Sibelius® 7
Using the ManuScript language
Written by Jonathan Finn, James Larcombe, Yasir Assam, Simon Whiteside, Mike Copperwhite, Paul Walmsley, Graham Westlake and Michael Eastwood, with contributions from Andrew Davis and Daniel Spreadbury.

See the About Sibelius dialog for a full list of the software development team and other credits.

We would like to thank all those (too numerous to list) who have provided helpful comments and suggestions for Sibelius and its documentation.

Legal Notices
This product is subject to the terms and conditions of a software license agreement.

This guide is copyrighted © 2011 by Avid Technology Inc., (hereafter “Avid”), with all rights reserved. Under copyright laws, this guide may not be duplicated in whole or in part without the written consent of Avid.

Avid, Sibelius and Scorch are registered trademarks of Avid Technology, Inc. in the USA, UK and other countries. All other trademarks are acknowledged as the property of their respective owners.

Product features, specifications, system requirements, and availability are subject to change without notice.

Document Feedback
We are always looking for ways to improve our documentation. If you have comments, corrections, or suggestions regarding our documentation, please email us at docs@sibelius.com.
Contents

Introduction ............................................................................................................................................................................. 5

Tutorial ........................................................................................................................................................................................................ 7
Edit Plug-ins ............................................................................................................................................................................. 8
Loops ......................................................................................................................................................................................... 13
Objects ..................................................................................................................................................................................... 15
Representation of a score ...................................................................................................................................................... 17
The “for each” loop .............................................................................................................................................................. 19
Indirection, sparse arrays and user properties .................................................................................................................... 21
Dialog editor .......................................................................................................................................................................... 25
Debugging your plug-ins ...................................................................................................................................................... 29
Storing and retrieving preferences .................................................................................................................................... 30

Reference .................................................................................................................................................................................................... 35
Syntax ........................................................................................................................................................................................... 36
Expressions ............................................................................................................................................................................... 38
Operators ................................................................................................................................................................................... 40

Object Reference ........................................................................................................................................................................ 41
Hierarchy of objects ................................................................................................................................................................. 42
All objects ................................................................................................................................................................................... 43
Accessibility ............................................................................................................................................................................... 45
Bar ............................................................................................................................................................................................ 46
Barline ........................................................................................................................................................................................ 52
Barlines ....................................................................................................................................................................................... 53
BarObject ................................................................................................................................................................................... 54
BarRest ......................................................................................................................................................................................... 57
Bracket ...................................................................................................................................................................................... 58
BracketsAndBraces ............................................................................................................................................................... 59
Clef ........................................................................................................................................................................................... 60
Comment ..................................................................................................................................................................................... 61
ComponentList ........................................................................................................................................................................ 62
Component ............................................................................................................................................................................... 63
DateTime ................................................................................................................................................................................... 64
Dictionary ................................................................................................................................................................................ 65
DocumentSetup ........................................................................................................................................................................ 66
DynamicPartCollection .......................................................................................................................................................... 68
DynamicPart .............................................................................................................................................................................. 69
EngravingRules ....................................................................................................................................................................... 70
File ......................................................................................................................................................................................... 72
Folder ....................................................................................................................................................................................... 73
GuitarFrame ............................................................................................................................................................................ 74
GuitarScaleDiagram ............................................................................................................................................................. 77
HitPointList ............................................................................................................................................................................ 78
HitPoint .................................................................................................................................................................................... 79
InstrumentChange ................................................................................................................................................................. 80
Global constants

What’s new in Sibelius 7
Introduction

ManuScript is a simple, music-based programming language developed to write plug-ins for the Sibelius music processor. The name was invented by Ben Sloman, a friend of Ben Finn’s.

It is based on Simkin, an embedded scripting language developed by Simon Whiteside, and has been extended by him and the rest of the Sibelius team ever since. (Simkin is a spooky pet name for Simon sometimes found in Victorian novels.) For more information on Simkin, and additional help on the language and syntax, go to the Simkin website at www.simkin.co.uk.

Rationale

In adding a plug-in language to Sibelius we were trying to address several different issues:

• Music notation is complex and infinitely extensible, so some users will sometimes want to add to a music notation program to make it cope with these new extensions.

• It is useful to allow frequently repeated operations (e.g. opening a MIDI file and saving it as a score) to be automated, using a system of scripts or macros.

• Certain more complex techniques used in composing or arranging music can be partly automated, but there are too many to include as standard features in Sibelius.

There were several conditions that we wanted to meet in deciding what language to use:

• The language had to be simple, as we want normal users (not just seasoned programmers) to be able to use it.

• We wanted plug-ins to be usable on any computer, as the use of computers running both Windows and Mac OS X is widespread in the music world.

• We wanted the tools to program in the language to be supplied with Sibelius.

• We wanted musical concepts (pitch, notes, bars) to be easily expressed in the language.

• We wanted programs to be able to talk to Sibelius easily (to insert and retrieve information from scores).

• We wanted simple dialog boxes and other user interface elements to be easily programmed.

C/C++, the world’s “standard” programming language(s), were unsuitable as they are not easy for the non-specialist to use, they would need a separate compiler, and you would have to recompile for each different platform you wanted to support (and thus create multiple versions of each plug-in).

The language Java was more promising as it is relatively simple and can run on any platform without recompilation. However, we would still need to supply a compiler for people to use, and we could not express musical concepts in Java as directly as we could with a new language.

So we decided to create our own language that is interpreted so it can run on different platforms, integrated into Sibelius without any need for separate tools, and can be extended with new musical concepts at any time.

The ManuScript language that resulted is very simple. The syntax and many of the concepts will be familiar to programmers of C/C++ or Java. Built into the language are musical concepts (Score, Staff, Bar, Clef, NoteRest) that are instantly comprehensible.

Technical support

Since the ManuScript language is more the province of our programmers than our technical support team (who are not, in the main, programmers), we can’t provide detailed technical help on it, any more than Oracle will help you with Java programming. This document and the sample plug-ins should give you a good idea of how to do some simple programming fairly quickly.

We would welcome any useful plug-ins you write – email them to daniel.spreadbury@avid.com and we may put them on our web site; if we want to distribute the plug-in with Sibelius itself, we’ll pay you for it.

Mailing list for plug-in developers

There is a growing community of plug-in developers working with ManuScript, and they can be an invaluable source of help when writing new plug-ins. To subscribe, go to http://avid-listsrv1.avid.com/mailman/listinfo/plugin-dev.
Tutorial
Edit Plug-ins

A simple plug-in

Let's start a simple plug-in. You are assumed to have some basic experience of programming (e.g. in BASIC or C), so you're already familiar with ideas like variables, loops and so on.

- Start Sibelius.
- Choose File › Plug-ins › Edit Plug-ins. The following dialog appears:

![Edit Plug-ins dialog]

- Now click New.

![New Manuscript Plug-in dialog]

- You are asked to type the internal name of your plug-in (used as the plug-in's filename), the name that should appear on the menu and the name of the category in which the plug-in should appear, which will determine which ribbon tab it appears on.
- The reason for having two separate names for plug-ins is that filenames may be no longer than 31 characters on Macs running Mac OS 9 (which is only significant if you intend your plug-in to be used with versions of Sibelius prior to Sibelius 4), but the menu names can be as long as you like.
- Type Test as the internal name, Test plug-in as the menu name and Tests as the category name, then click OK.
- You'll see Test (user copy) added to the list in the Edit Plug-ins dialog under a new Tests branch of the tree view. Click Close. This shows the folder in which the plug-in is located (Tests, which Sibelius has created for you), the filename of the plug-in (minus the standard .plg file extension), and (user copy) tells you that this plug-in is located in your user application data folder, not the Sibelius program folder or application package itself.
- If you look in the Home › Plug-ins gallery again you'll see a Tests category, with a Test plug-in underneath it.
Choose Home > Plug-ins > Tests > Test and the plug-in will run. You may first be prompted that you cannot undo plug-ins, in which case click Yes to continue (and you may wish to switch on the Don’t say this again option so that you’re not bothered by this warning in future.) What does our new Test plug-in do? It just pops up a dialog which says Test (whenever you start a new plug-in, Sibelius automatically generates in a one-line program to do this). You’ll also notice a window appear with a button that says Stop Plug-in, which appears whenever you run any plug-in, and which can be useful if you need to get out of a plug-in you’re working on that is (say) trapped in an infinite loop.

Click OK on the dialog and the plug-in stops.

Three types of information
Let’s look at what’s in the plug-in so far. Choose File > Plug-ins > Edit Plug-ins again, then select Tests/Test (user copy) from the list and click Edit (or simply double-click the plug-in’s name to edit it). You’ll see a dialog showing the three types of information that can make up a plug-in:

- **Methods**: similar to procedures, functions or routines in some other languages.
- **Dialogs**: the layout of any special dialogs you design for your plug-in.
- **Data**: variables whose value is remembered between running the plug-in. You can only store strings in these variables, so they’re useful for things like user-visible strings that can be displayed when the plug-in runs. For a more sophisticated approach to global variables, ManuScript provides custom user properties for all objects – see User properties on page 22.

Methods
The actual program consists of the methods. As you can see, plug-ins normally have at least two methods, which are created automatically for you when you create a new plug-in:

- **Initialize**: this method is called automatically whenever you start up Sibelius. Normally it does nothing more than add the name of the plug-in to the Plug-ins menu, although if you look at some of the supplied plug-ins you’ll notice that it’s sometimes also used to set default values for data variables.
- **Run**: this is called when you run the plug-in, you’ll be startled to hear (it’s like main() in C/C++ and Java). In other words, when you choose Home > Plug-ins > Tests > Test, the plug-in’s Run method is called. If you write any other methods, you have to call them from the Run method – otherwise how can they ever do anything?
Click on **Run**, then click **Edit** (or you can just double-click **Run** to edit it). This shows a dialog where you can edit the **Run** method:

![Menu/Script Method](image)

In the top field you can edit the name; in the next field you can edit the parameters (i.e. variables where values passed to the method are stored); and below is the code itself:

```
Sibelius.MessageBox("Test");
```

This calls a method **MessageBox** which pops up the dialog box that says **Test** when you run the plug-in. Notice that the method name is followed by a list of parameters in parentheses. In this case there’s only one parameter: because it’s a string (i.e. text) it’s in double quotes. Notice also that the statement ends in a semicolon, as in C/C++ and Java. If you forget to type a semicolon, you’ll get an error when the plug-in runs.

What is the role of the word **Sibelius** in **Sibelius.MessageBox**? In fact it’s a variable representing the Sibelius program; the statement is telling Sibelius to pop up the message box (C++ and Java programmers will recognize that this variable refers to an “object”). If this hurts your brain, we’ll go into it later.

### Editing the code

Now try amending the code slightly. You can edit the code just like in a word processor, using the mouse and arrow keys, and you can also also use **Ctrl-X/C/V** or **⌘X/C/V** for cut, copy and paste respectively. If you right-click (Windows) or **Control-click** (Mac) you get a menu with these basic editing operations on them too.

Change the code to this:

```
let x = 1;
    x = x + 1;
Sibelius.MessageBox("1 + 1 = " & x);
```

You can check this makes sense (or, at least, some kind of sense) by clicking the **Check syntax** button. If there are any blatant mistakes (e.g. missing semicolons) you’ll be told where they are.

Then close the dialogs by clicking **OK, OK** again then **Close**. Run your amended plug-in from the **Plug-ins** menu and a message box with the answer **1 + 1 = 2** should appear.

How does it work? The first two lines should be obvious. The last line uses **&** to stick two strings together. You can’t use + as this works only for numbers (if you try it in the example above, you’ll get an interesting answer!).

One pitfall: try changing the second line to:

```
x += 1;
```

then click **Check syntax**. You’ll get an error: this syntax (and the syntax **x++**) is allowed in various languages but not in ManuScript. You have to do `x = x+1`.

### Where plug-ins are stored

Plug-ins supplied with Sibelius are stored in folders buried deep within the Sibelius program folder on Windows, and inside the application package (or “bundle”) on Mac. It is not intended that end users should add extra plug-ins to these locations.
themselves, as we have provided a per-user location for plug-ins to be installed instead. When you create a new plug-in or edit an existing one, the new or modified plug-in will be saved into the per-user location (rather than modifying or adding to the plug-ins in the program folder or bundle):

- On Windows, additional plug-ins are stored at `C:\Users\username\AppData\Roaming\Avid\Sibelius 7\Plugins`.
- On Mac, additional plug-ins are stored in subfolders at `/Users/username/Library/Application Support/Avid/Sibelius 7/Plugins`.

This is worth knowing if you want to give a plug-in to someone else. The plug-ins appear in subfolders which correspond to the categories in which they appear in the various **Plug-ins** galleries. The filename of the plug-in itself is the plug-in's internal name plus the `.plg` extension, e.g. `Test.plg`.

(Sibelius includes an automatic plug-in installer, which you can access via **File > Plug-ins > Install Plug-ins**. This makes it easy to download and install plug-ins from the Sibelius web site.)

### Line breaks and comments

As with C/C++ and Java, you can put new lines wherever you like (except in the middle of words), as long as you remember to put a semicolon after every statement. You can put several statements on one line, or put one statement on several lines.

You can add comments to your program, again like C/C++ and Java. Anything after `//` is ignored to the end of the line. Anything between `/*` and `*/` is ignored, whether just part of a line or several lines:

```markdown
// comment lasts to the end of the line
/* you can put
several lines of comments here */
```

For instance:

```plaintext
Sibelius.MessageBox("Hi!");  // print the active score
```

or:

```plaintext
Sibelius /* this contains the application */.MessageBox("Hi!");
```

### Variables

In ManuScript a variable can be any sequence of letters, digits or `_` (underscore), as long as it doesn't start with a digit.

A variable can contain an integer (whole number), a floating point number, a string (text) or an object (e.g. a note) – more about objects in a moment. Unlike most languages, in ManuScript a variable can contain any type of data – you don't have to declare what type you want. Thus you can store a number in a variable, then store some text instead, then an object. Try this:

```plaintext
x = 56; x = x+1;
Sibelius.MessageBox(x);       // prints '57' in a dialog box
x = "now this is text";       // the number it held is lost
Sibelius.MessageBox(x);       // prints 'now this is text' in a dialog
x = Sibelius.ActiveScore;     // now it contains a score
Sibelius.MessageBox(x);       // prints nothing in a dialog
```

Variables that are declared within a ManuScript method are local to that method; in other words, they cannot be used by other methods in the same plug-in. **Global Data** variables defined using the plug-in editor can be accessed by all methods in the plug-in, and their values are preserved over successive uses of the plug-in.

A quick aside about strings in ManuScript is in order at this point. Like many programming languages, ManuScript strings uses the back-slash `\` as an “escape character” to represent certain special things. To include a single quote character in your strings, use `\'`, and to include a new line you should use `\n`. Because of this, to include the backslash itself in a ManuScript string one has to write `\\`.

### Converting between numbers, text and objects

Notice that the method **MessageBox** is expecting to be sent some text to display. If you give it a number instead (as in the first call to **MessageBox** above) the number is converted to text. If you give it an object (such as a score), no text is produced.
Similarly, if a calculation is expecting a number but is given some text, the text will be converted to a number:

```plaintext
x = 1 + "1";                        // the + means numbers are expected
Sibelius.MessageBox(x);            // displays '2'
```

If the text doesn't start with a number (or if the variable contains an object instead of text), it is treated as 0:

```plaintext
x = 1 + "fred";
Sibelius.MessageBox(x);            // displays '1'
```
Loops

“for” and “while”
ManuScript has a while loop which repeatedly executes a block of code until a certain expression becomes True. Create a new plug-in called Potato. This is going to amuse one and all by writing the words of the well-known song “1 potato, 2 potato, 3 potato, 4”. Type in the following for the Run method of the new plug-in:

```manuscript
x = 1;
while (x<5)
{
    text = x & " potato,";
    Sibelius.MessageBox(text);
    x = x+1;
}
```

Run it. It should display “1 potato,” “2 potato,” “3 potato,” “4 potato,” which is a start, though annoyingly you have to click OK after each message.

The while statement is followed by a condition in ( ) parentheses, then a block of statements in { } braces (you don't need a semicolon after the final } brace). While the condition is true, the block is executed. Unlike some other languages, the braces are compulsory (you can't omit them if they only contain one statement). Moreover, each block must contain at least one statement. We did say that ManuScript was a simple language.

In this example you can see that we are testing the value of x at the start of the loop, and increasing the value at the end. This common construct could be expressed more concisely in ManuScript by using a for loop. The above example could also be written as follows:

```manuscript
for x = 1 to 5
{
    text = x & " potato,";
    Sibelius.MessageBox(text);
}
```

Here, the variable x is stepped from the first value (1) up to the end value (5), stopping one step before the final value. By default, the “step” used is 1, but we could have used (say) 2 by using the syntax for x = 1 to 5 step 2, which would then print only “1 potato” and “3 potato”!

Notice the use of & to add strings. Because a string is expected on either side, the value of x is turned into a string.

Notice also we’ve used the Tab key to indent the statements inside the loop. This is a good habit to get into as it makes the structure clearer. If you have loops inside loops you should indent the inner loops even more.

The if statement
Now we can add an if statement so that the last phrase is just “4,” not “4 potato”:

```manuscript
x = 1;
while (x<5)
{
    if(x=4)
    {
        text = x & ".";
    }
    else
    {
        text = x & " potato,";
    }
    Sibelius.MessageBox(text);
    x = x+1;
}
```
Tutorial

The rule for if takes the form if (condition) { statements}. You can also optionally add else { statements}, which is executed if the condition is false. As with while, the parentheses and braces are compulsory, though you can make the program shorter by putting braces on the same line as other statements:

```plaintext
x = 1;
while (x<5)
{
    if(x=4) {
        text = x & ".";
    } else {
        text = x & " potato,"
    }
    Sibelius.MessageBox(text);
    x = x+1;
}
```

The position of braces is entirely a matter of taste.

Now let's make this plug-in really cool. We can build up the four messages in a variable called text, and only display it at the end, saving valuable wear on your mouse button. We can also switch round the if and else blocks to show off the use of not. Finally, we return to the for syntax we looked at earlier.

```plaintext
text = "";                                // start with no text
for x = 1 to 5
{
    if (not(x=4)) {
        text = text & x & " potato, "; // add some text
    } else {
        text = text & x & ".";       // add no. 4
    }
}
Sibelius.MessageBox(text);                 // finally display it
```

Arithmetic

We've been using + without comment, so here's a complete list of the available arithmetic operators:

- a + b         add
- a - b         subtract
- a * b         multiply
- a / b         divide
- a % b         remainder
- -a            negate
- (a)           evaluate first

ManuScript evaluates operators strictly from left-to-right, unlike many other languages; so 2+3*4 evaluates to 20, not 14 as you might expect. To get the answer 14, you'd have to write 2+(3*4).

ManuScript also supports floating point numbers, so whereas in some early versions 3/2 would work out as 1, it now evaluates to 1.5. Conversion from floating point numbers to integers is achieved with the RoundUp(expr), RoundDown(expr) and Round(expr) functions, which can be applied to any expression.
Now we come to the neatest aspect of object-oriented languages like ManuScript, C++ or Java, which sets them apart from traditional languages like BASIC, Fortran and C. Variables in traditional languages can hold only certain types of data: integers, floating point numbers, strings and so on. Each type of data has particular operations you can do to it: numbers can be multiplied and divided, for instance; strings can be added together, converted to and from numbers, searched for in other strings, and so on. But if your program deals with more complex types of data, such as dates (which in principle you could compare using =, < and >, convert to and from strings, and even subtract) you are left to fend for yourself.

Object-oriented languages can deal with more complex types of data directly. Thus in the ManuScript language you can set a variable, let's say `thischord`, to be a chord in your score, and (say) add more notes to it:

```plaintext
thischord.AddNote(60);           // adds middle C (note no. 60)
thischord.AddNote(64);           // adds E (note no. 64)
```

If this seems magic, it's just analogous to the kind of things you can do to strings in BASIC, where there are very special operations which apply to text only:

```plaintext
A$ = "1"
A$ = A$ + " potato, ":           REM add strings
X = ASC(A$):                     REM get first letter code
```

In ManuScript you can set a variable to be a chord, a note in a chord, a bar, a staff or even a whole score, and do things to it. Why would you possibly want to set a variable to be a whole score? So you can save it or add an instrument to it, for instance.

**Objects in action**

We'll have a look at how music is represented in ManuScript in a moment, but for a little taster, let's plunge straight in and adapt `Potato` to create a score:

```plaintext
x = 1;
text = "";                                    // start with no text
while (x<5)
{
    if (not(x=4)) {
        text = text & x & " potato, ";           // add some text
    } else {
        text = text & x & ".";                   // add no. 4
    }
    x = x+1;
}
Sibelius.New();                               // create a new score
newscore = Sibelius.ActiveScore;              // put it in a variable
newscore.CreateInstrument("Piano");
staff = newscore.NthStaff(1);                 // get top staff
bar = staff.NthBar(1);                        // get bar 1 of this staff
bar.AddText(0,text,"Technique");            // use Technique text style
```

This creates a score with a Piano, and types our potato text in bar 1 as Technique text.

The code uses the period (.) several times, always in the form `variable.variable` or `variable.method()`. This shows that the variable before the period has to contain an object.

- If there's a variable name after the period, we're getting one of the object's sub-variables (called “fields” or “member variables” in some languages). For instance, if `n` is a variable containing a note, then `n.Pitch` is a number representing its MIDI pitch (e.g. 60 for middle C), and `n.Name` is a string describing its pitch (e.g. “C4” for middle C). The variables available for each type of object are listed later.
- If there's a method name after the period (followed by () parentheses), one of the methods allowed for this type of object is called. Typically a method called in this way will either change the object or return a value. For instance, if `s` is a variable con-
Tutorial

Taining a score, then `s.CreateInstrument("Flute")` adds a flute (changing the score), but `s.NthStaff(1)` returns a value, namely an object containing the first staff.

Let's look at the new code in detail. There is a pre-defined variable called Sibelius, which contains an object representing the Sibelius program itself. We've already seen the method `Sibelius.MessageBox()`. The method call `Sibelius.New()` tells Sibelius to create a new score. Now we want to do something to this score, so we have to put it in a variable.

Fortunately, when you create a new score it becomes active (i.e. its title bar highlights and any other scores become inactive), so we can just ask Sibelius for the active score and put it in a variable:

```
newscore = Sibelius.ActiveScore.
```

Then we can tell the score to create a Piano: `newscore.CreateInstrument("Piano")`. But to add some text to the score you have to understand how the layout is represented.
Representation of a score

A score is treated as a hierarchy: each score contains 0 or more staves; each staff contains bars (though every staff contains the same number of bars); and each bar contains “bar objects.” Clefs, text and chords are all different types of bar objects.

So to add a bar object (i.e. an object which belongs to a bar), such as some text, to a score: first you have to get which staff you want (and put it in a variable): \( \text{staff} = \text{newscore.NthStaff(1)} \); then you have to get which bar in that staff you want (and put it in a variable): \( \text{bar} = \text{staff.NthBar(1)} \); finally you tell the bar to add the text: \( \text{bar.AddText(0, text, "Technique")} \). You have to give the name (or index number – see Text styles on page 137) of the text style to use (and it has to be a staff text style, because we're adding the text to a staff).

Notice that bars and staves are numbered from 1 upwards; in the case of bars, this is irrespective of any bar number changes that are in the score, so the numbering is always unambiguous. In the case of staves, the top staff is no.1, and all staves are counted, even if they're hidden. Thus a particular staff has the same number wherever it appears in the score.

The AddText method for bars is documented later, but the first parameter it takes is a rhythmic position in the bar. Each note in a bar has a rhythmic position that indicates where it is (at the start, one quarter after the start, etc.), but the same is true for all other objects in bars. This shows where the object is attached to, which in the case of Technique text is also where the left hand side of the text goes. Thus to put our text at the start of the bar, we used the value 0. To put the text a quarter note after the start of the bar, use 256 (the units are 1024th notes, so a quarter is 256 units – but don't think about this too hard):

\[
\text{bar.AddText(256, text, "Technique")};
\]

To avoid having to use obscure numbers like 256 in your program, there are predefined variables representing different note values (which are listed later), so you could write:

\[
\text{bar.AddText(Quarter, text, "Technique")};
\]

or to be quaint you could use the British equivalent:

\[
\text{bar.AddText(Crotchet, text, "Technique")};
\]

For a dotted quarter, instead of using 384 you can use another predefined variable:

\[
\text{bar.AddText(DottedQuarter, text, "Technique")};
\]

or add two variables:

\[
\text{bar.AddText(Quarter+Eighth, text, "Technique")};
\]

This is much clearer than using numbers.

The system staff

As you know from using Sibelius, some objects don't apply to a single staff but to all staves. These include titles, tempo text, rehearsal marks and special barlines; you can tell they apply to all staves because (for instance) they get shown in all the instrumental parts.

All these objects are actually stored in a hidden staff, called the system staff. You can think of it as an invisible staff which is always above the other staves in a system. The system staff is divided into bars in the same way as the normal staves. So to add the title “Potato” to our score we'd need the following code in our plug-in:

\[
\begin{align*}
\text{sys} &= \text{newscore.SystemStaff}; & \text{// system staff is a variable} \\
\text{bar} &= \text{sys.NthBar(1)}; \\
\text{bar.AddText(0, "POTATO SONG", "Subtitle")};
\end{align*}
\]

As you can see, SystemStaff is a variable you can get directly from the score. Remember that you have to use a system text style (here I've used Subtitle) when putting text in a bar in the system staff. A staff text style like Technique won't work. Also, you have to specify a bar and position in the bar; this may seem slightly superfluous for text centered on the page as titles are (though in reality even this kind of page-aligned text is always attached to a bar), but for Tempo and Metronome mark text they are obviously required.
Representation of notes, rests, chords and other musical items

Sibelius represents rests, notes and chords in a consistent way. A rest has no noteheads, a note has 1 notehead and a chord has 2 or more noteheads. This introduces an extra hierarchy: most of the squiggles you see in a score are actually a special type of bar object that can contain even smaller things (namely, noteheads). There's no overall name for something which can be a rest, note or chord, so we've invented the pretty name NoteRest. A NoteRest with 0, 1 or 2 noteheads is what you normally call a rest, a note or a chord, respectively.

If \( n \) is a variable containing a NoteRest, there is a variable \( n.\text{NoteCount} \) which contains the number of notes, and \( n.\text{Duration} \) which is the note-value in 1/256ths of a quarter. You can also get \( n.\text{Highest} \) and \( n.\text{Lowest} \) which contain the highest and lowest notes (assuming \( n.\text{NoteCount} \) isn't 0). If you set \( \text{lownote} = n.\text{Lowest} \), you can then find out things about the lowest note, such as \( \text{lownote.Pitch} \) (a number) and \( \text{lownote.Name} \) (a string). Complete details about all these methods and variables may be found in the Reference section below.

Other musical objects, such as clefs, lines, lyrics and key signatures have corresponding objects in ManuScript, which again have various variables and methods available. For example, if you have a Line variable \( l \), then \( l.\text{EndPoint} \) gives the rhythmic position at which the line ends.
The “for each” loop

It’s a common requirement for a loop to do some operation to every staff in a score, or every bar in a staff, or every BarObject in a bar, or every note in a NoteRest. There are other more complex requirements which are still common, such as doing an operation to every BarObject in a score in chronological order, or to every BarObject in a multiple selection. ManuScript has a for each loop that can achieve each of these in a single statement.

The simplest form of for each is like this:

```plaintext
thisscore = Sibelius.ActiveScore;
for each s in thisscore                         // sets s to each staff in turn
{
    // ...do something with s
}
```

Here, since thisscore is a variable containing a score, the variable s is set to be each staff in thisscore in turn. This is because staves are the type of object at the next hierarchical level of objects (see Hierarchy of objects on page 42). For each staff in the score, the statements in {} braces are executed.

Score objects contain staves, as we have seen, but they can also contain a Selection object, e.g. if the user has selected a passage of music before running the plug-in. The Selection object is a special case: it is never returned by a for each loop, because there is only a single selection object; if you use the Selection object in a for each loop, by default it will return BarObjects (not Staves, Bars or anything else!).

Let’s take another example, this time for notes in a NoteRest:

```plaintext
noterest = bar.NthBarObject(1);
for each n in noterest                          // sets n to each note in turn
{
    Sibelius.MessageBox("Pitch is " & n.Name);
}
```

n is set to each note of the chord in turn, and its note name is displayed. This works because Notes are the next object down the hierarchy after NoteRests. If the NoteRest is, in fact, a rest (rather than a note or chord), the loop will never be executed – you don’t have to check this separately.

The same form of loop will get the bars from a staff or system staff, and the BarObjects from a bar. These loops are often nested, so you can, for instance, get several bars from several staves.

This first form of the for each loop got a sequence of objects from an object in the next level of the hierarchy of objects. The second form of the for each loop lets you skip levels of the hierarchy, by specifying what type of object you want to get. This saves a lot of nested loops:

```plaintext
thisscore = Sibelius.ActiveScore;
for each NoteRest n in thisscore
{
    n.AddNote(60);                                // add middle C
}
```

By specifying NoteRest after for each, Sibelius knows to produce each NoteRest in each bar in each staff in the score; otherwise it would just produce each staff in the score, because a Staff object is the type of object at the next hierarchical level of objects. The NoteRests are produced in a useful order, namely from the top to the bottom staff, then from left to right through the bars. This is chronological order. If you want a different order (say, all the NoteRests in the first bar in every staff, then all the NoteRests in the second bar in every staff, and so on) you’ll have to use nested loops.

So here’s some useful code that doubles every note in the score in octaves:
score = Sibelius.ActiveScore;
for each NoteRest chord in score
{
    if(not(chord.NoteCount = 0))         // ignore rests
    {
        note = chord.Highest;            // add above the top note
        chord.AddNote(note.Pitch+12);    // 12 is no. of half-steps (semitones)
    }
}

It could easily be amended to double in octaves only in certain bars or staves, only if the notes have a certain pitch or duration, and so on.

This kind of loop is also very useful in conjunction with the user’s current selection. This selection can be obtained from a variable containing a Score object as follows:

selection = score.Selection;

We can then test whether it’s a passage selection, and if so we can look at (say) all the bars in the selection by means of a **for each** loop:

if (selection.IsPassage)
{
    for each Bar b in selection
    {
        // do something with this bar
        ...
    }
}

Be aware that you can not add or remove items from bars during iterating. The example of adding notes to chords above is fine because you are modifying an existing item (in this case a NoteRest), but it’s not safe to add or remove entire items, and if you try to do so, your plug-in will abort with an error. However, it’s very useful to add or remove items from bars, so you need to do that in a separate **for** loop, after first collecting the items you want to operate on into a ManuScript array, something like this:

num = 0;
for each obj in selection
{
    if (IsObject(obj))
    {
        n = "obj" & num;
        @n = obj;
        num = num + 1;
    }
}
selection.Clear();
for i = 0 to num
{
    n = "obj" & i;
    obj = @n; // get an object from the pseudo array
    obj.Select();
}

The @n in this example is the array. To find out more about arrays, read on.
Indirection, sparse arrays and user properties

Indirection

If you put the @ character before a string variable name, then the value of the variable is used as the name of a variable or method. For instance:

```manuscript
var="Name";
x = @var;                   // sets x to the contents of the variable Name
mymethod="Show";
@mymethod();                // calls the method Show
```

This has many advanced uses, though if taken to excess it can cause the brain to hurt. For instance, you can use @ to simulate "unlimited" arrays. If `name` is a variable containing the string "x1", then `@name` is equivalent to using the variable `x1` directly. Thus:

```manuscript
i = 10;
name = "x" & i;
@name = 0;
```

sets variable `x10` to 0. The last two lines are equivalent to `x[i] = 0;` in the C language. This has many uses; however, you'll also want to consider using the built-in arrays (and hash tables), which are documented below.

Sparse arrays

The method described above can be used to create “fake” arrays through indirection, though this is a little fiddly. ManuScript also provides Javascript-style sparse arrays, which can store anything that can be stored in a ManuScript variable, including references to objects. Like a variable, storing a reference to an object in a sparse array will preserve the lifetime of that object (because objects are reference counted), but the underlying object in Sibelius may become invalid if (say) a Score is modified.

To create a sparse array in ManuScript, use the built-in method `CreateSparseArray(a1, a2, a3, a4...an)`. You can create an empty array simply by passing in no variables to the `CreateSparseArray` method.

Sparse arrays provide a read/write variable called `Length` that returns or sets the length of the array: when you set `Length` to a number greater than the present size of the array, the array is padded with null values; if you set `Length` to a number smaller than the present size of the array, any values beyond this number are removed.

To push one or more values to the end of the array, use the method `Push(a1, a2, ... an)`. To remove and return the last element of an array, use the method `Pop()`.

An example of how to use a sparse array:

```manuscript
array = CreateSparseArray(4,5,6);
array[10] = 19; // creates 11th element of array, intervening elements are null
array.Length = 20; // extends array to 20 elements, new elements are all null
```

Sparse arrays by their nature may not have values in every array element. To return a new sparse array containing only the populated indices of the original sparse array (i.e. those that are not null), use the array's `ValidIndices` variable. For example, using the above sparse array:

```manuscript
array2 = array.ValidIndices; // will contain values 0, 1, 2, 10 and 19
return array[array2[0]]; // returns the first populated element of array
```

You can compare two sparse arrays for equality, e.g.:

```manuscript
if (array = array2) {
  // do something
}
```

To access the end of an array, it's convenient to use negative indices; e.g. `array[-1]` returns the last element, `array[-2]` returns the penultimate element, and so on. It's not possible to access elements before the start of the array, so if you do e.g. `array[-100]` on a six element array, you will get `array[0]` returned.
Some things to remember when using sparse arrays:

- Sparse arrays use a zero-based index.
- Elements that have not been initialised are null, and do not cause an error when referenced.
- Assigning to an index beyond the current length increases the \texttt{Length} to one greater than the index assigned to.
- If an array contains references to objects, whether the arrays are equal or not depends on the implementation of equality for those objects.

\textbf{User properties}

All ManuScript objects other than those listed below, including objects created by Sibelius, can have user properties attached to them, allowing for convenient storage of extra data, encapsulation of several items of data within a single object, and returning more than one value from a method, among other things.

To create a new user property, use the following syntax:

\begin{verbatim}
object._property:property_name = value;
\end{verbatim}

where \texttt{object} is the name of the object, \texttt{property\_name} is the desired user property name, and \texttt{value} is the value to be assigned to the new user property. User properties are read/write and can be accessed as \texttt{object.property\_name}.

To get a sparse array containing the names of all the user properties belonging to an object, you can do e.g.:

\begin{verbatim}
names = object._propertyNames;
\end{verbatim}

Here is an example of creating a user property:

\begin{verbatim}
nr = bar.NoteRest;
nr._property:original = true;
if (nr.original = true) {
    // do something
}
\end{verbatim}

Some things to remember when using user properties:

- If you attempt to get or set a user property that has not yet been created, your plug-in will exit with a run-time error.
- To check whether or not a user property has been created without causing a run-time error, use the notation \texttt{object._property:property\_name}, which will be null if no matching user property has been created yet.
- User properties cannot be created or accessed for normal data types (e.g. strings, integers, etc.), the global \texttt{Sibelius} object, old-style ManuScript arrays created by \texttt{CreateArray()}, old-style hashes created by \texttt{CreateHash()}, and null.
- User properties that conflict with an existing property name cannot be accessed as \texttt{object.property\_name} (though they can be accessed using the \texttt{._property:} notation).
- User properties belong to a particular ManuScript object and disappear when that object’s lifetime ends. To stop an object dying, you can (for example) store it in a sparse array, but be aware that its contents may become invalid if (say) the underlying score changes.

\textbf{Dictionary}

\texttt{Dictionary} is a programmer extensible object, simply allowing the use of user properties as above with convenient construction. It also has methods allowing the use of arbitrarily named user properties, and can also have methods in plug-ins attached to it allowing the creation of encapsulated user objects (i.e. objects with variables and methods attached to them).

To create a dictionary, use the built-in function \texttt{CreateDictionary(name1, value1, name2, value2, ... nameN, valueN)}. This creates a dictionary containing user properties called \texttt{name1}, \texttt{name2}, \texttt{nameN} with values \texttt{value1}, \texttt{value2}, \texttt{valueN} respectively.

A dictionary can contain named data items (like a \texttt{struct} in languages like C++), or data that is indexed by string, so that you can use strings to look items up within it.

The values in a dictionary can be accessed using square bracket notation, so you can use a dictionary like a hash table, e.g.:

\begin{verbatim}
test = CreateDictionary("fruit",apple,"vegetable",potato);
test["fruit"] = banana;
test["meat"] = lamb;
\end{verbatim}
You can even put other objects, e.g. sparse arrays, inside dictionaries, e.g.

test2 = CreateDictionary("fruit", CreateSparseArray(apple, banana, orange));

You can access the user properties within a dictionary using the .property: notation, e.g.:

return test2._property:fruit;

which would return the array specified above. Even more direct, you can access user properties in a dictionary as if they were variables or methods, like this:

test2.fruit;

which would also return the array specified above. You can also return more than one value from any ManuScript method using a dictionary, e.g.:

getChord()
value = CreateDictionary("a", aNote, "b", anotherNote);
return value;

//… in another method somewhere
chord = getChord();
trace(chord.a);
trace(chord.b);

which returns two values, a and b, which you can access via e.g. chord.a and chord.b.

You can compare two dictionaries for equality, e.g.:

if (test2 = test3) {
  // do something
}

Whether or not dictionaries containing objects evaluate as equal depends on the implementation of equality for those objects.

If you're comfortable with programming in general, you may find it useful to be able to add methods to dictionaries, particularly if you are writing code designed to act as a library for other methods or plug-ins to call. Writing code in this way provides a degree of encapsulation and can make it easy for client code to use your library.

To add a method to a dictionary, call the dictionary's SetMethod() method, e.g.:

pluginmethod "(obj,x,y) {
  // a method that does something to obj
}"

test4 = CreateDictionary();
test4.doSomething(3,4); // call pluginmethod within the current plug-in, passing in
  // test4 (obj in the method above) and 3 (x in the method
  // above) and 4 (y in the method above)

In the example above, doSomething is the name of the method belonging to the dictionary, Self tells the plug-in that the method is defined in the same plug-in, and pluginmethod is the name of a method elsewhere in the plug-in (shown at the top of the example).

To return a sparse array containing the names of the methods belonging to a dictionary, use the dictionary's GetMethodNames() method. You can also check the existence of a particular method using the dictionary's MethodExists() method. Use the dictionary's CallMethod() method to call a specific method, where the name of the method is the first parameter, and any parameters to be passed to the specified method follow. For example:

array = test4.GetMethodNames(); // create sparse array containing method names
first_method_name = array[0]; // sets first_method_name to name of first method
methodfound = test4.MethodExists("doSomething"); // returns True in this case;
test4.CallMethod("doSomething",5,6);

Everything you put into a dictionary is a user property, so all of the methods outlined in User properties above can be used on data in dictionaries too.
Using user properties as global variables

You can store `SparseArray` and `Dictionary` objects, and indeed any other object, as user properties of the Plugin object itself. In the example below, `Self` is the object that corresponds to the running plug-in, and a user property `globalData` is assigned to the plug-in, containing a Dictionary:

```javascript
Self._property:globalData = CreateDictionary(1,2,3,4);
// globalData and Self.globalData can be used interchangeably
trace(globalData);
trace(Self.globalData);
```

User properties assigned to the plug-in are persistent between invocations. Take care to ensure that these user properties are created before you attempt to use them, otherwise your plug-in will abort with a run-time error. Using the `_property:property_name` syntax never causes run-time errors, but direct references to `property_name` force a runtime error if `property_name` hasn’t been created yet.

The example below shows how to test the existence of a specific user property, `globalCounter`, initialise it to 0 if it is not found, then increment it by 1 every time the plug-in runs:

```javascript
// Test the persistence of user properties
if (Self._property:globalCounter = null) {
    Self._property:globalCounter = 0;
}
globalCounter = globalCounter + 1;
// this number increases by one every time the plugin is run
trace(globalCounter);
trace(Self.globalCounter);
```

If you store a reference to a musical object in a user property that is assigned to the plug-in, there is an increased danger of that reference becoming invalid due to the score being closed or edited, etc. Use the `IsValid()` method to validate such data before using it.

User properties of plug-ins will be inaccessible (except by using the `_property:property_name` syntax) if there is an existing global variable of the same name.

Watch out for recursive cycles!

Be careful not to create recursive cycles using arrays, user properties and dictionaries. When you use, say, an array in a dictionary, you are not creating a copy of the array or its values, but a reference to the original array: dictionaries and arrays are objects, not values. As a result, you could write something where an array contains a dictionary that itself refers to the original array: this will lead to Sibelius crashing. So be careful!

Other things to look out for

The `Parallel 5ths and 8ves` plug-in illustrates having several methods in a plug-in, which we haven't needed so far. The `Proof-read` plug-in illustrates that one plug-in can call another – it doesn't do much itself except call the `CheckPizzicato`, `CheckSuspectClefs`, `CheckRepeats` and `CheckHarpPedaling` plug-ins. Thus you can build up meta-plug-ins that use libraries of others. Cool!

(You object-oriented programmers should be informed that this works because, of course, each plug-in is an object with the same powers as the objects in a score, so each one can use the methods and variables of the others.)
Dialog editor

For more complicated plug-ins than the ones we've been looking at so far, it can be useful to prompt the user for various settings and options. This may be achieved by using ManuScript's simple built-in dialog editor. Dialogs can be created in the same way as methods and data variables in the plug-in editor.

Showing a dialog in a plug-in

To show a dialog from a ManuScript method, we use the built-in call

```
Sibelius.ShowDialog(dialogName, Self);
```

where `dialogName` is the name of the dialog we wish to show, and `Self` is a “special” variable referring to this plug-in (telling Sibelius to whom the dialog belongs). Control will only be returned to the method once the dialog has been closed by the user.

Creating or editing a dialog

To create a new dialog, choose the Dialog radio button at the bottom of the window that lists methods, data and dialogs, and click Add. To edit an existing dialog, select it from the Dialogs list box at the top right-hand corner of the window, and click Edit.

The dialog form will then appear, along with a long thin “palette” of available controls, as follows:

- Radio button
- Checkbox
- Button
- Static text
- Editable text
- Combo box
- List box
- Group box

To create a new control, simply drag and drop it from the palette onto the dialog.

Dialog properties

With no controls selected, either double-click on a blank part of the dialog (or right-click, and then choose Properties) to access the dialog's Properties dialog, which allows you to specify:

- **Name**: the value of `dialogName` for the `Sibelius.ShowDialog()` method call (see Showing a dialog in a plug-in above).
- **Title**: the name of the dialog as it appears in its title bar.
- **Size**: the Width and Height (measured in somewhat arbitrary dialog units); you can also set the size of the dialog by resizing it directly when editing it.
- **Position**: the X and Y position that the dialog should open at by default.

Laying out controls

The dialog editor includes a number of simple options for producing a pleasing layout:

- To select a control, either double-click on it or hit Tab to select the next control in the creation order (Shift-Tab selects the previous control).
- To nudge a selected control, simply use the arrow keys.
- To align controls, select them using Ctrl-click (Windows) or ⌘-click (Mac), then use e.g. Ctrl+← or ⌘← to align all of the selected controls with the left-hand edge of the left-most control, or Ctrl+↑ or ⌘↑ to align all of the selected controls with the top edge of the top-most control.
- To space controls evenly, select them using Ctrl-click (Windows) or ⌘-click (Mac), then use e.g. Ctrl+Shift+Alt+← or ⌘+← to space the controls evenly in the distance between the top edge of the top-most and the bottom edge of the bottom-most controls, or Ctrl+Shift+Alt+↑ or ⌘+↑ to space the controls evenly in the distance between the left-hand edge of the
left-most and the right-hand edge of the right-most controls. Once controls are spaced evenly, you can increase or decrease the space between them proportionally by typing **Ctrl+Shift+Alt+↑/↓/←/→** or **⌥±⌘↑/↓/←/→** as appropriate.

You can optionally display a grid to aid with alignment. Right-click on a blank part of the dialog and choose **Grid** from the context menu to see a dialog with settings for the grid:

![Grid Settings](image)

Switch on **Show grid** to show the grid in the editor. Choose between **Dots** or **Lines**, and specify the **Opacity** of the grid display by adjusting the slider. Switch on **Snap to grid** to enable control snapping as you drag them with the mouse. Although a control that you nudge with the keyboard will not snap to the grid, one side of its selection outline will flash when it comes into alignment with the grid in either the horizontal or vertical directions.

**Undo and redo**

You can undo and redo everything you have done while editing a dialog using **Ctrl+Z** or **⌘Z** to undo and **Ctrl+Y** or **⌘Y** to redo.

**Testing your dialog**

To test your dialog within the editor, right-click a blank part of the dialog and choose **Test** from the context menu (or type the shortcut **Ctrl+T** or **⌘T**). To finish testing and return to the editor, simply hit **Esc**, or click any control whose properties are set to close the dialog (e.g. an **OK** or **Cancel** button, if you have created one).

**Saving your changes**

To save the changes to your dialog, simply click the close button in the dialog's title bar: if you have any unsaved changes, Sibelius will ask you whether or not to save the changes.

**Set Creation Order**

If you have done any programming in other languages that allow you to edit dialogs, you will probably be familiar with the concept of **tab order**, which refers to the order in which controls are given the focus when the user repeatedly hits the **Tab** key to cycle through them. ManuScript has a similar concept called **creation order**, so named because the order in which the controls in a dialog are created affects not only the tab order but also some other subtle things (including radio button grouping – see **Radio buttons** below).

To set the creation order of controls in your plug-in's dialog, right-click on a blank part of the dialog and choose **Set Creation Order** from the context menu. A special display appears overlaid on the controls in your dialog, like this:

![Change Dynamics](image)

To set the creation order, simply click on each control in order. If you make a mistake, hold down **Ctrl** or **⌘** and click on the last control whose order is correct to restart the sequence from that point, then release **Ctrl** or **⌘** and resume clicking on the remaining controls. Once you're done, hit **Esc** to finish editing the creation order.
Control properties
Every control that you create also has a Properties dialog, which can be accessed by double-clicking a selected control (or by right-clicking and choosing Properties from the context menu, or by typing the shortcut Ctrl+Return or ⌘ Return). The dialog for a radio button control, for example, is shown on the right.

With a control selected, the properties window varies depending on the type of the control, but most of the options are common to all controls, and these are as follows:

- **ID**: an internal string that identifies the control; Sibelius generates this for you automatically, but you can change if you like
- **Text**: the text appearing in the control
- **Position (X, Y)**: where the control appears in the dialog, in coordinates relative to the top left-hand corner
- **Size (width, height)**: the size of the control
- **Variable storing control's value**: the ManuScript Data variable that will correspond to the value of this control when the plug-in is run
- **Method called when clicked**: the ManuScript method that should be called whenever the user clicks on this control (leave blank if you don't need to know about users clicking on the control)
- **Click closes dialog**: select this option if you want the dialog to be closed whenever the user clicks on this control. The additional options Returning True / False specify the value that the Sibelius.ShowDialog method should return when the window is closed in this way.
- **Give this control focus**: select this option if the “input focus” should be given to this control when the dialog is opened, i.e. if this should be the control to which the user's keyboard applies when the dialog is opened. Mainly useful for editable text controls.

Other options vary according to the type of control selected.

Combo boxes and list boxes
Combo boxes and list boxes have an additional property; you can set a variable from which the control's list of values should be taken. Like the value storing the control's current value, this should be a global Data variable. However, in this instance they have a rather special format, to specify a list of strings rather than simply a single string. Look at the variable _ComboItems in Add String Fingering for an example – it looks like this:

```csharp
_comboItems
{
  "1",
  "2",
  "3",
  "4",
  "1 and 3",
  "2 and 4"
}
```

Radio buttons
Radio buttons also have an additional property that allows one to specify groups of radio buttons in plug-in dialogs. When the user clicks on a radio button in a group, only the other radio buttons belonging to that groups are deselected; any others in the dialog are left as they are. This is extremely useful for more complicated dialogs.

To specify a radio group, pick one control from each group that represents the first button of the group, and for these controls ensure that the checkbox Start a new radio group is selected in the control's Properties dialog. Then set the creation order of the controls (see Set Creation Order above). A radio button group is defined as being all the radio buttons created between two buttons that have the Start a new radio group flag set (or between one of these buttons and the end of the dialog). So to make the radio groups work properly, ensure that each group is created sequentially in order, with the button at the start of the group created first, and then all the rest of the radios in that group. To finish, click the Set Creation Order menu item again to deactivate this mode.
**Static text**

Static text controls additionally allow you to determine whether the text should be aligned to the **Left** (useful for explanatory text) or to the **Right** (useful for text associated with a specific control to its right, such as an edit control, checkbox or combo box).

**Buttons**

In most plug-in dialogs, you will want the **OK** button to be the default button for the dialog, such that if the user hits **Return** or **Enter** on their keyboard, the dialog is confirmed, and closes. Likewise, you will want the **Cancel** button to respond to the user hitting **Esc** on their keyboard, closing the dialog without making any changes.

For **OK** buttons, or other buttons that should confirm the dialog, switch on the **Default button for dialog** checkbox in the button's **Properties** dialog. Each dialog should only have one default button. You will also normally set **Click closes dialog, returning** to **True**. Depending on the other controls in your dialog, you may additionally want to check **Give this control focus**; if you have one or more edit controls in the dialog, you should probably set **Give this control focus** on the first of those controls instead.

**Cancel** buttons, by contrast, should normally only have **Click closes dialog, returning** set to **False**.
Debugging your plug-ins

When developing any computer program, it’s all too easy to introduce minor (and not so minor!) mistakes, or bugs. ManuScript performs its own internal error checking at all times, and you’ll find that if you try to access a non-existent method or variable on an object, or make a syntax error, or attempt to add or remove bars or items from bars while iterating over them, the plug-in will throw an error and open the plug-in editor window at the line that generated the error.

As ManuScript is a simple, lightweight system, there is no special purpose debugger, but there are a handful of tools provided to help you debug your plug-ins.

**Undo**

One good technique for finding problems in your plug-ins is to set Sibelius's undo buffer to a very small size, or to disable it altogether (by dragging the slider on the Other page of File > Preferences to its leftmost position). In the unlikely event that ManuScript does not throw an error when you perform an illegal operation (e.g. adding or deleting an object while iterating over a bar), reducing the undo buffer to its smallest possible size will expose the problem right away – though be warned, the result of such a problem may well be that Sibelius will crash.

**Plug-in Trace Window**

The trace window can be shown by choosing Plug-ins > Plug-in Trace Window. A special ManuScript command, `trace(string)`, will print the specified string in the trace window. This is useful to keep an eye on what your plug-in is doing at particular points. These commands can then be removed when you’ve finished debugging. Another useful feature of the trace window is function call tracing. When this is turned on, the log will show which functions are being called by plug-ins.

One potential pitfall with the `trace(string)` approach to debugging is that the built-in hash table and array objects discussed earlier aren’t strings, and so can’t be output to the trace window. To avoid this problem, both of these objects have a corresponding method called `WriteToString()`, which returns a string representing the whole structure of the array or hash at that point. So we could trace the current value of an array variable as follows:

```
trace("array variable = " & array.WriteToString());
```

**Checking the validity of objects**

One of the common problems that you might encounter when writing complex plug-ins is that the object you are trying to operate on is no longer valid (e.g. it has already been deleted). You can enable error checking – either for all objects, or for individual objects – that will cause your plug-in to throw an error if an object is no longer valid.

To enable error checking, use the ManuScript command `ValidationChecking(enable[, object1[, object2]...])`, and set the Boolean parameter `enable` to `true`. If `enable` is the only parameter, validation checking is enabled for all types of objects, and all plug-ins. If you supply one or more `object` parameters (e.g. `Tuplet`, `Score`, `BarObject`, etc.), only those objects will be checked, and only in the currently running plug-in. You should ensure `ValidationChecking` is set to `false` before you give your plug-ins to anybody else to use.

You can also use the special method `IsValid()` to determine whether an object is valid: it will return `false` if the object in question no longer exists. `GetValidation返回搜狐` returns an empty string if there is no error, or returns a string if an error has occurred, so you can do e.g. `trace(GetValidationError(score))` to trace any validation error returned by a Score object to the trace window.

**Stopping the plug-in**

If you want to force your plug-in to stop on a particular error condition, use the method `StopPlugin([message])`, which will stop your plug-in, display the optional `message` in an alert box, and open the plug-in editor at the line of code reached.

You can also use `ExitPlugin()`, which exits the plug-in cleanly without dropping into the plug-in editor.
Storing and retrieving preferences

In Sibelius 4 or later, you can use Preferences.plg, contributed by Hans-Christoph Wirth, to store and retrieve user-set preferences for your plug-ins.

How does it work?
Preferences.plg stores its data in a text file in the user's application data folder. Strings are accessed as associated pairs of a key (the name of the string) and a value (the contents of the string). The value can also be an array of strings, if required.

Initializing the database

```plaintext
errorcode = Open(pluginname, featureset);
```

Open the library and lock for exclusive access by the calling plug-in. The calling plug-in is identified with the string plugin-name. It is recommended that this string equals the unique Sibelius menu name of the calling plug-in.

Parameter featureset is the version of the feature set requested by the calling plug-in. The version of the feature set is currently 020000. Each library release shows in its initial dialog a list of supported feature sets. The call to Open() will fail and show a user message if you request an unsupported feature set. If you should want to prevent this user information (and probably setup your own information dialog), use CheckFeatureSet() first.

After Open() the scope is undefined, such that you can access only global variables until the first call to SwitchScope().

Return value: Open() returns zero or a positive value on success. A negative result indicates that there was a fatal error and the database has not been opened.

-2 other error
-1 library does not support requested feature set
0 no common preferences database found
1 no preferences found for current plug-in
2 preferences for current plug-in loaded

In case of errors (e.g. if the database file is unreadable), Open() offers the user an option to recover from the error condition. Only if this fails too will an error code be returned to the calling plug-in.

```plaintext
errorcode = CheckFeatureSet(featureset);
```

Check silently if the library supports the requested feature set.

Return value: CheckFeatureSet() returns zero or a positive value on success. A negative value indicates that the requested feature set is not supported by this version.

```plaintext
errorcode = Close();
```

Release the exclusive access lock to the library. If there were any changes since the last call to Open() or Write(), dump the data changes back to disk (probably creating a new score, if there was none present).

Return value: Close() returns zero or a positive value on success. A negative result indicates that there was a fatal error and the database has not been written.

```plaintext
errorcode = CloseWithoutWrite();
```

Release the exclusive access lock to the library, discarding any changes performed since last call to Open() or Write().

Return value: CloseWithoutWrite() returns zero or a positive value on success. A negative result indicates that there was a fatal error, namely that the database was not open at the moment.

```plaintext
errorcode = Write(dirty);
```

Force writing the data back to disk immediately. Keep library locked and open. If dirty equals 0, the write only takes place if the data has been changed. If dirty is positive, the common preferences score is unconditionally forced to be rewritten from scratch.

Return value: Write() returns zero or a positive value on success. A negative result indicates that there was a fatal error and the database has not been written.
**Accessing data**

```plaintext
index = SetKey(keyname, value);
```

Store a string value `value` under the name `keyname` in the database, overwriting any previously stored keys or arrays of the same `keyname`.

If `keyname` has been declared as a local key, the key is stored within the current scope and does not affect similar keys in other scopes. It is an error to call `SetKey()` for local keys if the scope is undefined.

Return value: `SetKey()` returns zero or a positive value on success, and a negative value upon error.

```plaintext
errorcode = SetArray(keyname, array, size);
```

Store a array `array` of strings under the name `keyname` in the database, overwriting any previously stored keys or arrays of the same `keyname`. `size` specifies the number of elements in the array. A `size` of `-1` is replaced with the natural size of the array, i.e., `array.NumChildren`.

If `keyname` has been declared as a local key, the array is stored within the current scope and does not affect similar keys in other scopes. It is an error to call `SetArray()` for local keys if the scope is undefined.

Return value: `SetArray()` returns zero or a positive value on success, and a negative value upon error.

```plaintext
value = GetKey(keyname);
```

Retrieve the value of key `keyname` from the database. It is an error to call `GetKey()` on an identifier which had been stored the last time using `SetArray()`. For local keys, the value is retrieved from the current scope which must not be undefined.

Return value: The value of the key or `Preferences.VOID` if no key of that name found.

```plaintext
size = GetArray(keyname, myarray);
```

Retrieve the string array stored under name `keyname` from the database. It is an error to call `GetArray()` on an identifier which has been stored the last time by `SetKey()`. For local arrays, the value is retrieved from the current scope which must not be undefined.

You must ensure before the call that `myarray` is of ManuScript’s array type (i.e., created with `CreateArray()`).

Return value: `size` equals the number of retrieved elements or `-1` if the array was not found. Note that `size` might be smaller than `myarray.NumChildren`, because there is currently no way to reduce the size of an already defined array.

```plaintext
size = GetListOfIds(myarray);
```

Fill the array `myarray` with a list of all known Ids in the current score (or in the global scope, if undefined). Before you call this method, ensure that `myarray` is of ManuScript’s array type (i.e. created with `CreateArray()`).

Return value: returns the size of the list, which might be smaller than the natural size of the array, `myarray.NumChildren`.

```plaintext
index = UnsetId(keyname);
```

Erase the contents stored with an identifier (there is no distinction between keys and arrays here). If the key is declared as local, it is erased only from the local scope which must not be undefined.

Return value: The return value is zero or positive if the key has been unset. A negative return value means that a key of that name has not been found (which is not an error condition).

```plaintext
RemoveId(keyname);
```

Erase all contents stored in the database with an identifier (there is no distinction between keys and arrays here). If the key is declared as local, it is erased from all local scopes.

Return value: The return value is always zero.

```plaintext
RemoveAllIds();
```

Erase everything related to the current plug-in.

Return value: the return value is always zero.
**Commands for local variables**

```plaintext
errorcode = DeclareIdAsLocal(keyname);
```

Declare an identifier as a local key. Subsequent calls to `Set...` and `Get...` operations will be performed in the scope which is set at that time. The local state is stored in the database and can be undone by a call to `DeclareIdAsGlobal` or `RemoveId`.

Return value: Non-negative on success, negative on error.

```plaintext
size = GetListOfLocalIds(myarray);
```

Fill the array `myarray` with a list of all Ids declared as local. Before you call this method, ensure that `myarray` is of ManuScript's array type (i.e. created with `CreateArray()`).

Return value: Returns the size of the list, which might be smaller than the natural size of the array, `myarray.NumChildren`.

```plaintext
errorcode = SwitchScope(scopename);
```

Select scope `scopename`. If scope `scopename` has never been selected before, it is newly created and initialized with no local variables. Subsequent `Set...` and `Get...` operations for keys declared as local will be performed in scope `scopename`, while access to global keys is still possible.

The call `SwitchScope("")` selects the undefined scope which does not allow access of any local variables.

Return value: Non-negative on success, negative on error.

```plaintext
errorcode = RemoveScope();
```

Erase all local keys and arrays from the current scope and delete the current scope from the list of known scopes. It is an error to call `RemoveScope()` if the current scope is undefined. After the call, the database remains in the undefined scope.

```plaintext
errorcode = RemoveAllScopes();
```

Erase all local keys and arrays from all scopes and delete all scopes from the list of known scopes. After the call, the database remains in the undefined scope. Note that this call does retain the information which Ids are local (see `DeclareIdAsLocal()`).

Return value: Non-negative on success.

```plaintext
string = GetCurrentScope();
```

Retrieve the name of the currently active scope, or the empty string if the database is in undefined scope.

Return value: Returns a string.

```plaintext
size = GetListOfScopes(myarray);
```

Fill the array `myarray` with a list of all known scope names. You must ensure before the call that `myarray` is of ManuScript's array type (i.e., created with `CreateArray()`).

Return value: Returns the size of the list, which might be smaller than the natural size of the array, `myarray.NumChildren`.

**Miscellaneous**

```plaintext
Trace(tracelevel);
```

Select level of tracing for the library. Useful levels are: 0 for no trace, 10 for sparse trace, 20 for medium trace, 30 for full trace. This command can also be run when the library is not open, to specify the tracing level for the `Open()` call itself.

```plaintext
TraceData();
```

Writes a full dump of the data stored currently in `ThisData` array to the trace window. This is the full data belonging to the current plug-in. `TraceData()` always traces the data, regardless of the current trace level selected.

```plaintext
filename = GetFilename();
```

Return the full filename of the preferences database (including path).

Editor();
Invoke the interactive plug-in editor. This method must not be called while the database is open. Direct calls to `Editor()` from plug-ins are deprecated, since the end-user of your plug-in will probably not expect to be able to edit (and destroy) the saved preferences of all plug-ins at this stage.

**Basic example**

Suppose you have a plug-in called `myplugin` and would like to save some dialog settings in a preferences file such that these settings are persistent over several Sibelius sessions and computer reboots. Your dialog may contain two checkboxes and a list box. Let `DialogDontAskAgain` and `DialogSpeedMode` be the global variables holding the status of the checkboxes, respectively, and let `DialogJobList` hold the contents of the list box item.

The work with the database can be reduced to four steps:

1. **Open the database and retrieve initial data.** At begin of your plug-in, e.g. right at top of your `Run()` method, you have to add some code to initialize the database. You probably also want to initialize your global keys based on the information currently stored in the database. See below for a detailed example. (Depending on your program, you might have to define `prefOpen` as a global variable in order to prevent trying to access an unopened database in future.)

```cpp
// At first define hard coded plug-in defaults, in case that the plug-in
// is called for the first time. If anything else fails, these defaults
// will be in effect.
DialogDontAskAgain = 0;
DialogSpeedMode = 0;
DialogJobList = CreateArray();
DialogJobList[0] = "first job";
DialogJobList[1] = "second job";

// Attempt to open the database
prefOpen = Preferences.Open( "myplugin", "020000" );
if( prefOpen >= 0 ) {
    // Database successfully opened. So we can try to load the
    // information stored last time.
    // It’s a good idea to work with a private version scheme, in order
    // to avoid problems in the future when the plug-in is developed
    // further, but the database still contains the old keys. In our
    // example, we propose that the above mentioned keys are present
    // if "version" key is present and has a value of "1".
    version = Preferences.GetKey( "Version" );
    switch( version ) {
        case( "1" ) {
            // Now overwrite the above set defaults with the information stored
            // in the database.
            DialogDontAskAgain = Preferences.GetKey( "DontAskAgain" );
            DialogSpeedMode = Preferences.GetKey( "SpeedMode" );
            Preferences.GetArray( "JobList", DialogJobList );
        }
        default {
            // handle other versions/unset version gracefully here ...
        }
    }
}
```
2. Work with the data. After the initialization step, you can and should work with global variables `DialogDontAskAgain`, `DialogSpeedMode`, and `DialogJobList` as you are used to: read from them to base control flow decisions on their setting, write to them (mostly from within your own dialogs) to set new user preferences.

3. Write data back to the database. To make any changes persistent, you must tell the database the new values to be written to the hard disk. See below for a detailed example. According to taste, you can execute these lines each time the settings are changed, or only once, at the end of your plug-in.

   ```c
   if( prefOpen >= 0 ) {
       Preferences.SetKey( "Version", "1" );
       Preferences.SetKey( "DontAskAgain", DialogDontAskAgain );
       Preferences.SetKey( "SpeedMode", DialogSpeedMode );
       Preferences.SetArray( "JobList", DialogJobList, -1 );
   }
   ```

4. Close the database. In any case, you must release the lock to the library on exit of your plug-in. This writes data actually back to disk, and enables other plug-ins to access the shared database later. To do this, use:

   ```c
   Preferences.Close();
   ```
Syntax

Here is an informal run-down of the syntax of ManuScript.

A method consists of a list of statements of the following kinds:

Block

\[
\{ \text{statements} \}
\]

\[
\{ \ a = 4; \\
\}
\]

While

\[
\text{while} \ \{ \ \text{expression} \} \ \text{block}
\]

\[
\text{while} \ (i < 3) \ { \\
\quad \text{Sibelius.MessageBox}(i); \\
\quad i = i + 1;
\}
\]

Switch

\[
\text{switch} \ (\text{test-expression}) \ { \\
\quad \text{case} \ (\text{case-expression-1}) \ \text{block} \\
\qquad \text{[case} \ (\text{case-expression-2}) \ \text{block} \ \\
\qquad \text{]} \\
\quad \text{[default} \ \text{block} \]
\]

The switch statement consists of a “test” expression, multiple case statements and an optional default statement. If the value of test-expression matches one of the case-expressions, then the statement block following the matching case statement will be executed. If none of the case statements match, then the statement block following the default statement will be executed. For example:

\[
\text{switch} \ (\text{note.Accidental}) \ { \\
\quad \text{case} \ (\text{DoubleSharp}) \ { \\
\quad \quad \text{Sibelius.MessageBox}(\text{"Double sharp"}); \\
\quad \} \\
\quad \text{case} \ (\text{DoubleFlat}) \ { \\
\quad \quad \text{Sibelius.MessageBox}(\text{"Double flat"}); \\
\quad \} \\
\quad \text{default} \ { \\
\quad \quad \text{Sibelius.MessageBox}(\text{"No double"}); \\
\quad \}
\}
\]

if else

\[
\text{if} \ (\text{expression}) \ \text{block} \ [ \text{else} \ \text{block}]
\]

\[
\text{if} \ (\text{found}) \ { \\
\quad \text{Application.ShowFindResults}(\text{found}); \\
\} \ \text{else} \ { \\
\quad \text{Application.NotFindResults}();
\}
\]
for each

```plaintext
for each variable in expression
block
```

This sets variable to each of the sub-objects within the object given by the expression.

Normally there is only one type of sub-object that the object can contain. For instance, a
NoteRest (such as a chord) can only contain Note objects. However, if more than one type of
sub-object is possible you can specify the type:

```plaintext
for each Type variable in expression
block
```
e.g.

```plaintext
for each NoteRest n in thisstaff {
    n.AddNote(60);          // add middle C
}
```

for

```plaintext
for variable = value to value / step value /
block
```

The variable is stepped from the first value up to or down to the end value by the step value.
It stops one step before the final value.

So, for example:

```plaintext
for x=1 to note.NoteCount {
    ...
}
```

works correctly.

assignment

```plaintext
variable = expression;
```
e.g.

```plaintext
value = value + 1;
```
or

```plaintext
variable . variable = expression;
```
e.g.

```plaintext
Question.CurrentAnswer=True;
```

method call

```plaintext
variable . identifier (comma-separated expressions);
```
e.g.

```plaintext
thisbar.AddText(0,"Mozart","text.system.composer");
```

self method call

```plaintext
identifier (comma-separated expressions);
```

Calls a method in this plug-in, e.g.

```plaintext
CheckIntervals();
```

return

```plaintext
return expression;
```

Returns a value from a plug-in method, given by the expression. If a method doesn't con-
tain a return statement, then a “null” value is returned (either the number zero, an empty
string, or the null object described below).
Expressions

Here are the operators, literals and other beasts you’re allowed in expressions.

Self
This is a keyword referring to the plug-in owning the method. You can pass yourself to other methods, e.g.

```java
other.Introduce(Self);
```

null
This is a literal object meaning “nothing.”

Identifier
This is the name of a variable or method (letters, digits or underscore, not starting with a digit) you can precede the identifier with @ to provide indirection; the identifier is then taken to be a string variable whose value is used as the name of a variable or method.

member variable

```java
variable
```

This accesses a variable in another object.

integer
e.g. `1, 100, -1`

floating point number
e.g. `1.5, 3.15, -1.8`

string
Text in double quotes, e.g. "some text". For strings that are rendered by Sibelius as part of the score, i.e. the contents of some text object, there is a small but useful formatting language allowing one to specify how the text should appear. These "styled strings" contain commands to control the text style. All commands start and end with a backslash (`\`) The full list of available styling commands is as follows:

- `\n\n` New paragraph
- `\n\n` New line
- `\b\b` Bold on
- `\b\b` Bold off
- `\i\i` Italic on
- `\i\i` Italic off
- `\u\u` Underline on
- `\u\u` Underline off
- `\fArial Black\` Font change to Arial Black (for example)
- `\fArial Black\` Font change to Arial Black (for example)
- `\fArial Black\` Font change to Arial Black (for example)
- `\text.character.musictext` Character style change to Music text (for example)
- `\f_` Font change to text style’s default font, including removing any active character styles
- `\s123\` Size change to 123 (units are 1/32nds of a space, not points)
- `\v\v` Vertical scale in percent
- `\h\h` Horizontal scale in percent
- `\t\t` Tracking (absolute) in 1/32nds of a space
- `\p\` Baseline adjustment: use normal, sub (for subscript) or super (for superscript)
- `\$keyword\` Substitutes a string from the Score Info dialog (see below)

A consequence of this syntax is that backslashes themselves are represented by `\`, to avoid conflicting with the above commands.

The substitution command `\$keyword\` supports the following keywords: Title, Composer, Arranger, Lyricist, MoreInfo, Artist, Copyright, Publisher and PartName. Each of these correspond to a field in the File > Score Info dialog.
not

```
expression not
```
Logically negates an expression, e.g.
```
not (x=0)
```

and

```
expression and expression
```
Logical and, e.g.
```
FoxFound and BadgerFound
```

or

```
expression or expression
```
Logical or, e.g.
```
FoxFound or BadgerFound
```

equality

```
expression = expression
```
Equality test, e.g.
```
Name="Clock"
```

subtract

```
expression – expression
```
Subtraction, e.g.
```
12-1
```

add

```
expression + expression
```
Addition, e.g.
```
12+1
```

minus

```
-expression
```
Inversion, e.g.
```
-1
```

concatenation

```
expression & expression
```
Add two strings, e.g.
```
Name = "Fred" & "Bloggs"; // 'Fred Bloggs'
```

You can’t use + as this would attempt to add two numbers, and sometimes succeed (!). For instance:
```
x = "2" + "2"; // same as x = 4
```

subexpression

```
(expression)
```
For grouping expressions and enforcing precedence, e.g.
```
(4+1)*5
```

method call

```
variable.identifier (comma-separated expressions)
```

```
e.g.
x = monkey.CountBananas();
```

self method call

```
Identifier (comma-separated expressions)
```

Calls a method in this plug-in, e.g.
```
x = CountBananas();
```
Operators

Condition operators
You can put any expressions in parentheses after an `if` or `while` statement, but typically they will contain conditions such as `=` and `<`. The available conditions are very simple:

- `a = b` equals (for numbers, text or objects)
- `a < b` less than (for numbers)
- `a > b` greater than (for numbers)
- `c and d` both are true
- `c or d` either are true
- `not c` inverts a condition, e.g. `not (x=4)`
- `<=` less than or equal to
- `>=` greater than or equal to
- `!=` not equal to

Note that you use `=` to compare for equality, not `==` as found in C/C++ and Java.

Arithmetic

- `a + b` add
- `a - b` subtract
- `a * b` multiply
- `a / b` divide
- `a % b` remainder
- `-a` negate
- `(a)` evaluate first

ManuScript will evaluate expressions from left to right, so that `2+3*4` is 20, not 14 as you might expect. To avoid problems with evaluation order, use parentheses to make the order of evaluation explicit. To get the answer 14, you’d have to write `2+(3*4)`.

ManuScript also now supports floating point numbers, so whereas in previous versions `3/2` would work out as 1, it now evaluates to 1.5. Conversion from floating point numbers to integers is achieved with the `RoundUp(expr)`, `RoundDown(expr)` and `Round(expr)` functions, which can be applied to any expression.
Object Reference
**Object Reference**

**Hierarchy of objects**

```
Sibelius object
  ↓
Score
  ↓
  VersionHistory
  ↓
  Version
  ↓
  VersionComment

  ↓
DynamicPartCollection

  ↓
DynamicPart

  ↓
Selection

  ↓
Stave (including the SystemStave)

  ↓
Bar

  ↓
Text, Clef, Line, TimeSignature, KeySignature,
  Highlight, Lyric, Barline, Tuple, GuitarFrame,
  GuitarScaleDiagram, Comment,
  NoteRest (these are all BarObjects)

  ↓
Note (in NoteRests only)
```
Methods

AddToPluginsMenu("menu text", "function name")

Adds a new menu item to the Plug-ins menu. When the menu item is selected the given function is called. This is normally only used by plug-ins themselves. This method may only be called once per plug-in (i.e. each plug-in may only add one item to the Plug-ins menu); subsequent method calls will be ignored.

Asc(expression)

Returns the ASCII value of a given character (the expression should be a string of length 1).

CharAt(expression, position)

Returns the character from the expression at the given (zero-based) position, e.g. CharAt("Potato", 3) would give “a.”

Chr(expression)

Returns a character (as a string) with the given ASCII value. This can be used for inserting double quotes (”) into strings with Chr(34).

CreateArray()

Returns a new array object.

CreateHash()

Returns a new hash-table object.

GetValidationError(object)

Returns the validation error, if any, of the specified object. Useful to pass validation errors to the plug-in trace window.

ExitPlugin()

Exits the plug-in cleanly without dropping into the plug-in editor.

IsObject(expression)

Returns 1 (or True) if expression evaluates to an object rather than a null, boolean, string, or any number. (Not to be confused with the IsPassage variable of Selection objects!)

IsValid(object)

Returns 1 (or True) if the object is valid, returns 0 (or False) if the object no longer exists (i.e. has been deleted).

JoinStrings(expression, delimiter)

Joins together (concatenates) an array of strings into a single string, separated by the string delimiter.

Length(expression)

Gives the number of characters in the value of the expression.

Round(expression)

Returns the nearest integer to the value of the expression, e.g. Round(1.5) would be “2” and Round(1.3) would be “1.”

RoundDown(expression)

Returns the nearest integer less than the value of the expression, e.g. RoundDown(1.5) would be “1.”

RoundUp(expression)

Returns the nearest integer greater than the value of the expression, e.g. RoundUp(1.5) would be “2.”

SplitString(expression, [delimiter, ]trimEmpty)

Splits a string into an array of strings, using the given delimiter. The delimiter can be a single character or a string containing several characters – for instance "..,*" would treat either a comma or full stop as a delimiter. The default delimiter is the space
character. If the trimEmpty parameter is True then this will ignore multiple delimiters (which would otherwise produce some empty strings in the array). The default value of trimEmpty is False.

```plaintext
s = ':a:b:c';
bits = SplitString(s, ':', false);
// bits[0] = ''; bits[1] = 'a'; bits[2] = 'b' ...

s = 'a b c';
bits = SplitString(s, ' ', true);
// bits[0] = 'a'; bits[1] = 'b' ...
```

StopPlugin([message])

Stops the plug-in, and shows the optional message in an alert box. Opens the plug-in editor at the line of code reached.

Substring(expression, start, [length])

This returns a substring of the expression starting from the given start position (zero-based) up to the end of the expression, e.g. Substring("Potato", 2) would give "tato". When used with the optional length parameter, Substring returns a substring of the of the expression starting from the given start position (zero-based) of the given length, e.g. Substring("Potato", 2, 2) would give "ta".

Trace(expression)

Sends a piece of text to be shown in the plug-in trace window, e.g. Trace("Here's a trace");

ValidationChecking(enable[, object1[, object2]...])

If enable is the only parameter, validation checking is enabled for all types of objects, and across all plug-ins. If you supply one or more object parameters (e.g. Tuple, Score, BarObject, etc.), only those objects will be checked, and only in the currently running plug-in. You should ensure ValidationChecking is set to false before you give your plug-ins to anybody else to use.

User properties

All objects (except for the Sibelius object, old-style ManuScript arrays created using CreateArray(), old-style ManuScript hashes created using CreateHash(), and null) can also have user properties assigned to them. See User properties on page 22 for more details.
Accessibility

Accessed from the Sibelius object.

Methods
None.

Variables
- **ScoreDescription**
  Returns true if Sibelius's built-in score description functionality is enabled (read/write).
Bar

A Bar contains BarObject objects.

for each variable in produces the BarObjects in the bar for each type variable in produces the BarObjects of the specified type in the bar

Methods

AddBarNumber(new bar number[, format[, extra_text[, prepend[, skip this bar]]]])

Adds a bar number change to the start of this bar. new bar number should be the desired external bar number. The optional format parameter takes one of the three pre-defined constants that define the bar number format; see Global constants on page 135. The optional extra_text parameter takes a string that will be added after the numeric part of the bar number, unless the optional boolean parameter prepend is True, in which case the extra_text is added before the numeric part of the bar number. If the optional skip this bar parameter is True, the bar number change is created with the Don't increment bar number option set. Returns the BarNumber object created.

AddChordSymbolFromPitches(position, pitches[, instrument style])

Adds a chord symbol from the given array of pitches at the specified position. The optional instrument style parameter operates the same as in the AddGuitarFrame method (see above). If the method is unable to create a chord symbol, the method returns null; otherwise it returns the GuitarFrame object created.

AddClef(pos, concert pitch clef[, transposed pitch clef])

Adds a clef to the staff at the specified position. concert pitch clef determines the clef style when Score is switched off; the optional transposed pitch clef parameter determines the clef style when this is switched on. Clef styles should be an identifier like “clef.treble”; for a complete list of available clef styles, see Clef styles on page 139. Alternatively you can give the name of a clef style, e.g. “Treble,” but bear in mind that this may not work in non-English versions of Sibelius. Returns the Clef object created.

AddComment(sr, text[, color[, maximized]])

Adds a comment at the specified sr position in the current bar, displaying the specified text. The optional color parameter allows you to specify the color of the comment that is created (if not specified, the comment is created with its default color), and the optional maximized Boolean parameter allows you to set the comment to be minimized (if not specified, the comment is created maximized by default). If you want to specify the maximized parameter without specifying a particular color, set color to -1.

AddCommentWithName(sr, text, username[, color[, maximized]])

Adds a comment that will display a given username at the specified sr position in the current bar, displaying the specified text. The optional color parameter allows you to specify the color of the comment that is created (if not specified, the comment is created with its default color), and the optional maximized Boolean parameter allows you to set the comment to be minimized (if not specified, the comment is created maximized by default). If you want to specify the maximized parameter without specifying a particular color, set color to -1.

AddGraphic(fileName, pos[, below staff[, x displacement[, y displacement[, size ratio]]]])

Adds a graphic above or below the bar at a given position. If below staff is True, Sibelius will position the graphic below the staff to which it is attached, otherwise it will go above (the default). You may additionally displace the graphic from its default position by specifying x- and y displacements. These should be expressed in millimeters, the latter defining an offset from the top or bottom line of the staff, as appropriate. By default, the graphic will be created 5mm away from the staff. To adjust the size of the graphic, you may set a floating point number for its size ratio. When set to 1.0 (the default), the graphic will be created with a height equal to that of the staff to which it is attached. A value of 0.5 would therefore halve its size, and 2.0 double it. The graphic may be rescaled to a maximum of five times the height of its parent staff. This function returns True if successful, otherwise False.
AddGraphicToBlankPage(file name, nth page, x offset, y offset[, size ratio])

Adds a graphic to a blank page belonging to the current bar. nth page specifies the particular blank page you would like the graphic to, starting from 1. The x offset and y offset parameters are floating point values relative to the size of the page the graphic is being added to. For example, an x offset of 0.0 would position the graphic at the very left of the page; 0.5 in the centre. You may specify the size of the graphic by specifying a value for size ratio. This defaults to 1.0, which has the same effect as creating a graphic in Sibelius manually using Create > Graphic. (As with AddGraphic, 0.5 would halve its size, and 2.0 double it.) The graphic may be rescaled to a maximum of five times its initial size. This function returns True if successful, otherwise False.

AddGuitarFrame(position, chord name[, instrument style[, fingerings]])

Adds a chord symbol for the given chord name to the bar. The optional instrument style parameter should refer to an existing instrument type that uses tab, and should be specified by identifier; see Instrument types on page 139. If instrument style is not specified, Sibelius will create a chord symbol that will optionally display a chord diagram using the default tab tuning associated with the instrument type used by the staff to which the chord symbol will be attached. The position is in 1/256th quarters from the start of the bar. The optional fingerings parameter gives the fingerings string to display above (or below) the guitar frame, if supplied. If the method is unable to create a chord symbol, the method returns null; otherwise it returns the GuitarFrame object created.

AddInstrumentChange(pos, styleID[, add_clef[, show_text[, text_label[, show_warning[, warning_label[, full_instrument_name[, short_instrument_name]]]]]]])

Adds an instrument change to the bar at the specified position. styleID is the string representing the instrument type to change to (see Instrument types on page 139 for a list). The optional boolean parameter add_clef, True if not specified, determines whether Sibelius will add a clef change at the same position as the instrument change if required (i.e. if the clef of the new instrument is different to that of the existing instrument). show_text is an optional boolean parameter, True if not specified, determining whether or not the text label attached to the instrument change should be created shown (the default) or hidden. text_label is an optional string parameter; if specified, Sibelius will use this string instead of the default string (the new instrument's long name). show_warning is an optional boolean parameter, True if not specified, determining whether or not Sibelius should create a text object (using the Instrument change staff text style) above the last note preceding the instrument change, announcing the instrument change and giving the player time to pick up the new instrument. warning_label is an optional string parameter; if specified, Sibelius will use this string instead of the default string (the word “To” followed by the new instrument's short name). You can also override the names Sibelius will give the instruments on subsequent systems. If a null string is passed to either full_instrument_name or short_instrument_name (or if the arguments are omitted), the instrument names will remain unchanged. Returns the InstrumentChange object created.

AddKeySignatureFromText(pos, key name, major key[, add double barline[, hidden[, one staff only]]])

Adds a key signature to the bar. The key signature is specified by text name, e.g. “Cb” or “C#”. The third parameter is a Boolean flag indicating if the key is major (or minor). Unless the fourth parameter is set to False, a double barline will ordinarily be created alongside the key signature change. You may additionally hide the key signature change by setting hidden to True, and make the change of key appear on the bar’s parent staff only with the one staff only flag. Returns the key signature object created.

AddKeySignature(pos, num sharps, major key[, add double barline[, hidden[, one staff only]]])

Adds a key signature to the bar. The key signature is specified by number of sharps (+1 to +7), flats (-1 to -7), no accidentals (0) or atonal (-8). The third parameter is a Boolean flag indicating if the key is major (or minor). Unless the fourth parameter is set to False, a double barline will ordinarily be created alongside the key signature change. You may additionally hide the key signature change by setting hidden to True, and make the change of key appear on the bar’s parent staff only with the one staff only flag. Returns the key signature object created.

AddLine(pos, duration, line style[, dx[, dy[, voicenumber[, hidden]]]])

Adds a line to the bar. The line style can be an identifier such as “line.staff.hairpin.crescendo” or a name, e.g. “Crescendo”. For a complete list of line style identifiers that can be used in any Sibelius score, see Line styles on page 138. Style identifiers are to be preferred to named line styles as they will work across all language versions of Sibelius. Returns the Line object created, which may be one of a number of types depending on the Line style used.
AddLiveTempoTapPoint (position)

Adds a Live Tempo tap point at the rhythmic position specified by position, in 1/256th quarters from the start of the bar.

AddLyric (position, duration, text[, syllable type[, number of notes, voicenum]])

This method adds a lyric to the bar. The position is in 1/256th quarters from the start of the bar, and the duration is in 1/256th quarter units. The two optional parameters allow you to specify whether the lyric is at the end of a word (value is “1”, and is the normal value) or at the start or middle of a word (value is “0”), and how many notes the lyric extends beneath (default value 1). You can also optionally specify the voice in which the lyric should be created; if voicenum is 0 or not specified, the lyric is created in all voices. Returns the LyricItem object created.

AddNote (pos, sounding pitch, duration[, tied[, voice[, diatonic pitch[, string number]]]])

Adds a note to staff, adding to an existing NoteRest if already at this position (in which case the duration is ignored); otherwise creates a new NoteRest. Will add a new bar if necessary at the end of the staff. The position is in 1/256th quarters from the start of the bar. The optional tied parameter should be True if you want the note to be tied. Voice 1 is assumed unless the optional voice parameter (with a value of 1, 2, 3 or 4) is specified. You can also set the diatonic pitch, i.e. the number of the “note name” to which this note corresponds, 7 per octave (35 = middle C, 36 = D, 37 = E and so on). If a diatonic pitch of zero is given, a suitable diatonic pitch will be calculated from the MIDI pitch. The optional string number parameter gives a string number for this note, which is only meaningful if the note is on a tablature stave. If this parameter is not supplied then a default string number is calculated based on the current tablature stave type and the guitar tab fingering options (specified on the Notes page of File Preferences). Returns the Note object created (to get the NoteRest containing the note, use Note.ParentNoteRest).

AddPageNumber ([blank page offset])

Creates and returns a page number change at the end of the bar. Note that – due to the nature of adding a page number change – a page break will also be created at the end of the bar. Therefore, the page number change will actually be placed at the start of the next bar. The desired properties of the page number change can be set by calling the appropriate methods on the Page Number Change object returned.

The blank page offset flag allows you to create page number changes on blank pages. If a bar object is followed by one or more blank pages, each blank page may also have a page number change of its own. If unspecified, the page number change will be created on the next available page (whether it contains music or not) after the bar, otherwise the user may specify a 1-based offset which refers to the n-th blank page after the bar itself.

AddRehearsalMark ([consecutive[, mark[, new prefix and suffix[, prefix[, suffix[, override defaults]]]]]])

Adds a rehearsal mark above the bar. If no parameters have been specified, the rehearsal mark will inherit the properties of the previous rehearsal mark in the score, incrementing accordingly. Optionally, the appearance of the rehearsal mark may be overridden. If consecutive is False, Sibelius will not continue the numbering of the new rehearsal marks consecutively, but allow the user to set a new mark. A mark may be expressed as a number of a string. For example both 5 and “e” are both valid and equivalent values. If new prefix and suffix is True, the values set for prefix and suffix will be applied to the new rehearsal mark. The final parameter, override defaults, is a Boolean defaulting to False whose purpose it is to mimic the behavior of the option with the same name in the Rehearsal Mark dialog in Sibelius.

AddSpecialBarline (barline type[, pos])

Adds a special barline to a given position in a bar; see Global constants on page 135. If no position has been specified, start repeat barlines will snap to the start of the bar by default. All other special barline types will snap to the end.

AddSymbol (pos, symbol index or name)

Adds a symbol to the bar at the specified position. If the second parameter is a number, this is taken to be an index into the global list of symbols, corresponding to the symbol’s position in the Create Symbol dialog in Sibelius (counting left-to-right, top-to-bottom from zero in the top-left hand corner). Some useful symbols have pre-defined constants; see Global constants on page 135. There are also constants defined for the start of each group in the Create Symbol dialog, so that to access the 8va symbol, for example, you can use the index OctaveSymbols + 2. It’s better to use indices rather than names, because the names will be different across the various language versions of Sibelius. Returns the Symbol object created, or null if no symbol can be added to the score.
AddText (pos, text, style[, voicenum])

Adds the text at the given position, using the given text style. A staff text style must be used for a normal staff, and a system text style for a system staff. The styles should be an identifier of the form "text.staff.expression"; for a complete list of text styles present in all scores, see Text styles on page 137. Alternatively you can give the name of a text style, eg. "Expression", but be aware that this may not work in non-English versions of Sibelius. You can also optionally specify the voice in which the lyric should be created; if voicenum is 0 or not specified, the text object is created in all voices. Returns the Text object created.

AddTextToBlankPage (xPos, yPos, text, style, pageOffset)

Adds the text at the given position, using the given text style. A blank page text style must be used; you cannot add staff text or system text to a blank page. style takes a style ID, using the form "text.blankpage.title"; for a complete list of text styles present in all scores, see Text styles on page 137. xPos and yPos are the absolute position on the page. pageOffset takes a positive number for a blank page following a special page break (the first blank page is 1), and negative for a blank page preceding the first bar of the score (the blank page immediately before the first bar is -1, the one before that -2, and so on). Returns the Text object created.

To add text to a blank page, first create the special page break using the Bar.BreakType variable, and set the number of blank pages using Bar.NumBlankPages or Bar.NumBlankPagesBefore. Then use Bar.AddTextToBlankPage.

AddTimeSignature (top, bottom, allow cautionary, rewrite music[, use symbol])

Returns an error string (which will be empty if there was no error) which if not empty should be shown to the user. The first two parameters are the top and bottom of the new time signature. The third tells Sibelius whether to display cautionary time signatures from this time signature. If rewrite music is True then all the bars after the inserted the time signature will be rewritten. You can also create common time and alla breve time signatures. If you're creating a time signature in 4/4 or 2/2, set use symbol to True and Sibelius will replace the numbers of the time signature with their symbolic equivalent.

AddTimeSignatureReturnObject (top, bottom, allow cautionary, rewrite music[, use symbol])

As above, but returns the time signature object created, or null if unsuccessful.

AddTuplet (pos, voice, left, right, unit[, style[, bracket[, fullDuration]]])

Adds a tuplet to a bar at a given position. The left and right parameters specify the ratio of the tuplet, e.g. 3 (left) in the time of 2 (right). The unit parameter specifies the note value (in 1/256th quarters) on which the tuplet should be based. For example, if you wish to create an eighth note (quaver) triplet group, you would use the value 128. The optional style and bracket parameters take one of the pre-defined constants that affect the visual appearance of the created tuplet; see Global constants on page 135. If fullDuration is true, the bracket of the tuplet will span the entire duration of the tuplet. Returns the Tuplet object created.

N.B.: If AddTuplet() has been given illegal parameters, it will not be able to create a valid Tuplet object. Therefore, you should test for inequality of the returned Tuplet object with null before attempting to use it.

Bar [array element]

Returns the nth item in the bar (counting from 0) e.g. Bar[0]

Clear ([voice number])

Clears a bar of all its items, leaving only a bar rest. If a particular voice number is specified, only the items in that voice will be removed.

ClearNotesAndModifiers ([voice number])

Clears a bar of all its notes, rests, tuplets and slurs, replacing them with a single bar rest. If a particular voice number is specified, only the items in that voice will be removed.

Delete ()

Deletes and removes an entire bar from a score. This, by definition, will affect all the staves in the score.
DeletePageNumber ([blank page offset])

Deletes the page number change at the end of the bar, or if there are one or more blank pages after the bar, any page number change that occurs on any of those blank pages. If blank page offset is unspecified, the page number change on the first page after the bar will be deleted.

GetClefAt (pos)

Returns a Clef object corresponding to the current clef at the specified rhythmic position.

GetKeySignatureAt (pos)

Returns a KeySignature object corresponding to the current clef at the specified rhythmic position.

GetInstrumentTypeAt (pos)

Returns an InstrumentType object representing the instrument type used by the bar at the specified rhythmic position.

GetPageNumber ([blank page offset])

Returns the page number change object at the end of the bar, or if the bar contains no page number change, null. As with AddPageNumber, you may get the page number change from any of the blank pages that follow the bar by specifying a valid blank page offset.

InsertBarRest (voice number[, rest type])

Inserts a bar rest into the bar, but only if the bar is void of any NoteRests (or an existing bar rest) using the same voice number. The optional rest type parameter allows you to specify the type of bar rest or repeat bar to be created, defined by the constants WholeBarRest (the default if rest type is not specified), BreveBarRest, OneBarRepeat, TwoBarRepeat and FourBarRepeat. Returns True if successful.

NthBarObject (n)

Returns the nth object in the bar, counting from 0.

RemoveLiveTempoTapPoint (position)

Removes a Live Tempo tap point at the rhythmic position specified by position, in 1/256th quarters from the start of the bar.

ResetSpaceAroundBar (above, below)

Does the equivalent of Layout  Reset Space Above Staff and/or Reset Space Below Staff for the given bar. Set above to True to reset the space above the staff, and below to True to reset the space below the staff.

Respace ()

Respaces the notes in this bar.

Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BarNumber</td>
<td>The bar number of this bar. This is the internal bar number, which always runs consecutively from 1 (read only).</td>
</tr>
<tr>
<td>BarObjectCount</td>
<td>The number of objects in the bar (read only).</td>
</tr>
<tr>
<td>BreakType</td>
<td>The break at the end of this bar, given by the constants MiddleOfSystem, EndOfSystem, MiddleOfPage, EndOfPage, NotEndOfSystem, EndOfSystemOrPage or SpecialPageBreak. To learn the correspondence between these constants and the menu in the Bars panel of the Properties window, see the discussion in Global constants on page 135. When you set the break at the end of a bar to be SpecialPageBreak, Sibelius will add one blank page after the break. You can then adjust the number of pages by setting the value of either Bar.NumBlankPages or Bar.NumBlankPagesBefore, or tell Sibelius to restart the music on the next left or right page with Bar.MusicRestartsOnPage.</td>
</tr>
<tr>
<td>ExternalBarNumber</td>
<td>This has been deprecated as of Sibelius 5, because it can only return a number, and bar numbers that appear in the score may now include text. Use ExternalBarNumberString instead.</td>
</tr>
</tbody>
</table>

50
Returns the external bar number of this bar, taking into account bar number changes in the score (read only). Note that you cannot pass this bar number to any of the other ManuScript accessors; they all operate with the internal bar number instead.

**ExternalBarNumberString**
The external bar number of this bar as a string, taking into account bar number changes and bar number format changes (read only). Note that you cannot pass this bar number to any of the other ManuScript accessors; they all operate with the internal bar number instead.

**InMultirest**
Returns one of four global constants describing if and/or where the bar falls in a multirest (read only). The constants are `NoMultirest`, `StartsMultirest`, `EndsMultirest` and `MidMultirest`; see Global constants on page 135.

**Length**
The rhythmic length (read only).

**MusicRestartsOnPage**
Tells Sibelius to restart the music on the next left or right page after a special page break, and can only be set if `BreakType` is `SpecialPageBreak`. This variable may be set to only two of the global special page break constants: `MusicRestartsOnNextLeftPage` or `MusicRestartsOnNextRightPage` (write only).

**NthBarInSystem**
Returns the position of the bar in the system, relative to the first bar on the system (bar 0) (read only).

**NumBlankPages**
The number of blank pages following the bar containing a special page break.

**NumBlankPagesBefore**
The number of blank pages preceding the bar containing a special page break. This value only has an effect if a special page break exists in bar 1.

**OnHiddenStave**
Returns `True` if the bar is currently hidden by way of `Hide Empty Staves` (read only).

**OnNthPage**
Returns the zero-based page number on which the bar occurs in the current part (read only).

**OnNthPageExternal**
Returns a string containing the external page number (i.e. the page number displayed in the score) of the page in which the bar occurs.

**OnNthSystem**
Returns the zero-based system number (relative to its parent page) in which the bar occurs (read only).

**ParentStaff**
The staff containing this bar (read only).

**SectionEnd**
Corresponds to the Section end checkbox on the Bars panel of Properties (read/write).

**Selected**
Returns `True` if the entire bar is currently selected (read only).

**SpecialPageBreakType**
Returns the type of the special page break; see the documentation for the Special page break types in Global constants on page 135 (read only).

**SplitMultirest**
When `True`, a multirest intersecting the bar in question will be split (read/write).

**Time**
The time at which the bar starts in the score in milliseconds (read only).
Object Reference

Barline

Accessed from a Barlines object.

Methods
None.

Variables

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BottomStave</td>
<td>Returns the Staff object at which the barline ends, relative to the current part.</td>
</tr>
<tr>
<td>BottomStaveNum</td>
<td>Returns the number of the bottom staff included in the barline, relative to the current part.</td>
</tr>
<tr>
<td>TopStave</td>
<td>Returns the Staff object at which the barline starts, relative to the current part.</td>
</tr>
<tr>
<td>TopStaveNum</td>
<td>Returns the number of the top staff included in the barline, relative to the current part.</td>
</tr>
</tbody>
</table>
Barlines

Accessed from a Score object. Corresponds to the barline groupings in the score.

for each barline in iterates through all the barlines in the list, e.g.

```csharp
s = Sibelius.ActiveScore;
barlines = s.Barlines;
for each barline in barlines {
    // do something with barlines here
}
```

Array access [int n] returns the nth barline in the list, or null if the barline does not exist.

Methods

**AddBarline**(top staff number, bottom staff number)

Creates a new bar line inclusively spanning the staff numbers (relative to the current part) supplied. Returns the new Barline object created, or null if it fails.

**ClearAll()**

Removes all the barlines from the score.

**DeleteNthBarline**(index)

Removes a given barline identified by index from the score.

Variables

**NumChildren**

Returns the number of unique barlines in the score (read only).
BarObjects include Clef, Line, NoteRest & Text objects. All the methods and variables below apply to all specific types of BarObject – they are listed here instead of separately for each type. (To object-oriented programmers, the NoteRest, Clef etc. types are derived from the base class BarObject.)

Methods

**Delete()**

Deletes an item from the score. This will completely remove text objects, clefs, lines etc. from the score; however, when a NoteRest is deleted, it will be converted into a rest of similar duration. To delete multiple items from a bar, see Deleting multiple objects from a bar below.

**Deselect()**

Removes the object from the selection list of the parent score. If the selection is currently a passage selection, it is first changed to a multiple selection before the object is deselected. Returns True if the object is successfully removed from the selection.

**FreezeMagneticLayoutPositions()**

Does the same as selecting an object and choosing Layout > Freeze Magnetic Layout Positions, i.e. explicitly sets the object’s Dx/Dy to the position produced by Magnetic Layout, then disables Magnetic Layout for that object.

**GetIsInVoice(voicenum)**

Returns True if the object is in the voicenum specified.

**GetPlayedOnNthPass(n)**

Returns True if the object is set to play back the n'th time.

**NextItem({voice, item type})**

Returns the next item in the parent bar of the current item, or null if no item exists. If no arguments have been supplied, the very next item in the bar will be returned, regardless of its voice number and item type. You may additionally specify the voice number of the object you’re looking for (1 to 4, or 0 for any voice number), and the item’s type. Note that an item will only be returned if it exists in the same bar as the source item. By way of example, to find the next crescendo line in voice 2, you would type something along the lines of: hairpin = item.NextItem(2, “CrescendoLine”);

**PreviousItem({voice, item type})**

As above, but searches backwards.

**RemoveVoice(voicenum)**

Removes the object from the specified voicenum, leaving the object in all remaining voices.

**ResetPosition([horizontal[, vertical]])**

Performs Layout > Reset Position on the object. If you supply no parameters, this method will reset both the horizontal and vertical position of the object. If either or both of the optional Boolean parameters horizontal or vertical is set to True, you can reset the position of the object either horizontally or vertically independently if required.

**ResetDesign()**

Performs Layout > Reset Design on the object.

**Select()**

Appends the object to the selection list of the parent score. A multiple selection consisting of any number of individual objects can be built up by repeatedly calling Select on each object you wish to add to the list. Note that calling Select on a BarObject will first clear any existing passage selection.

**SetAllVoices()**

Sets the object to be in all voices. This has no effect on some types of object, e.g. NoteRests.

**SetVoice(voicenum[, clear other voices])**
Sets the object to be in voice *voicenum*, optionally removing the object from all other voices if the Boolean parameter *clear other voices* is *True*.

**ShowInAll()**

Shows the object in the full score, and in all relevant parts; equivalent to `Edit » Hide or Show » Show In All`.

**ShowInParts()**

Hides the object in the full score, and shows it in all relevant parts; equivalent to `Edit » Hide or Show » Show In Parts`.

**ShowInScore()**

Hides the object in all relevant parts, and shows it in the full score; equivalent to `Edit » Hide or Show » Show In Score`.

**SetPlayedOnNthPass(n, do play)**

Tells Sibelius whether or not the object should play back the *n*th time.

### Variables

- **CanBeInMultipleVoices**
  Returns *True* if the object can be in more than one voice (read-only).
- **Color**
  The color of this BarObject (read/write). The color value is in 24-bit RGB format, with bits 0–7 representing blue, bits 8–15 green, bits 16–23 red and bits 24–31 ignored. Since ManuScript has no bitwise arithmetic, these values can be a little hard to manipulate; you may find the individual accessors for the red, green and blue components to be more useful (see below).
- **ColorAlpha**
  The alpha channel component of the color of this BarObject, in the range 0–255 (read/write).
- **ColorRed**
  The red component of the color of this BarObject, in the range 0–255 (read/write).
- **ColorGreen**
  The green component of the color of this BarObject, in the range 0–255 (read/write).
- **ColorBlue**
  The blue component of the color of this BarObject, in the range 0–255 (read/write).
- **CueSize**
  *True* if the object is cue-size in the current part or score, and *False* if the object is normal size (read/write).
- **CurrentTempo**
  Returns the tempo, in bpm, at the location of the object in the score (read only).
- **DrawOrder**
  Returns the layer at which the object is currently drawn. When used to set the layer of an object, values from 1 (meaning the bottom layer) to 32 (meaning the highest layer) can be used; 0 is a special value that tells Sibelius to use the default layer for that type of object (read/write).
- **Dx**
  The horizontal graphic offset of the object from the position implied by the *Position* field, in units of 1/32 spaces (read/write).
- **Dy**
  The vertical graphic offset of the object from the centre staff line, in units of 1/32 spaces, positive going upwards (read/write).
- **HasCustomDrawOrder**
  Returns *True* if the object is set to a layer other than its default layer (read only).
- **Hidden**
  *True* if the object is hidden in the current part or score, and *False* if the object is shown (read/write).
- **OnNthBlankPage**
  Returns 0 if the object occurs on a page of music, otherwise a number from 1 upwards indicating the *n*th blank page of the bar on which the object occurs (read only).
- **ParentBar**
  The Bar containing this BarObject (read only).
- **Position**
  Rhythmic position of the object in the bar (read only).
- **Selected**
  Returns *True* if the object is currently selected (read only).
- **Time**
  The time at which the object occurs in the score in milliseconds (read only).
Object Reference

**Type**
A string describing the type of object, e.g. "NoteRest," “Clef.” This is useful when hunting for a specific type of object in a bar. See [GuitarScaleDiagram type values](#) on page 157 for the possible values (read only).

**UsesMagneticLayout**
Returns True if the object is positioned by Magnetic Layout. Returns False if the object is set not to be taken into account by Magnetic Layout. To set whether or not an object should use Magnetic Layout, use one of the global constants AlwaysDodge (equivalent to Edit ➤ Magnetic Layout ➤ On), SuppressDodge (Edit ➤ Magnetic Layout ➤ Off) or DefaultDodge (Edit ➤ Magnetic Layout ➤ Default) (read/write).

**UsesMagneticLayoutSettingOverridden**
Returns True if the object has had its Magnetic Layout settings overridden; otherwise False.

**VoiceNumber**
Is 0 if the item belongs to more than one voice (a lot of items belong to more than one voice) and 1 to 4 for items that belong to voices 1 to 4 (read only).

**Voices**
Returns or sets Sibelius's internal bitfield that represents the voices to which an object belongs; useful for copying the voices used by a given object (read/write).

**Deleting multiple objects from a bar**
If you wish to delete multiple objects from a bar, you should first build up a list of items to delete, then iterate over the list deleting each object in turn. It is not sufficient to simply delete the objects from the bar as you iterate over them, as this may cause the iterator to go out of sync. Therefore, code to delete all tuplets from a bar should look something like this:

```plaintext
counter = 0;
for each Tuplet tup in bar {
    name = "tuplet" & counter;
    @name = tup;
    counter = counter + 1;
}

// Delete objects in reverse order
while(counter > 0) {
    counter = counter - 1;
    name = "tuplet" & counter;
    tup = @name;
    tup.Delete();
}
```
BarRest

Derived from a BarObject object.

Methods
None.

Variables

PauseType
Returns the type of fermata (pause), if any, on the bar rest. Returns one of the constants PauseTypeNone (0), PauseTypeSquare (1), PauseTypeRound (2), PauseTypeTriangular (3) (read/write).

RestType
Returns the type of bar rest via one of the constants WholeBarRest (0), BreveBarRest (1), OneBarRepeat (2), TwoBarRepeat (3), FourBarRepeat (4) (read only). To create a bar rest of a particular type, use bar.InsertBarRest() (see above).
Bracket

Accessed from a BracketsAndBraces object.

**Methods**
None.

**Variables**

| BottomStave | Returns the Staff object at which the bracket ends, relative to the current part. |
| BottomStaveNum | Returns the number of the bottom staff included in the bracket, relative to the current part. |
| BracketType | Returns the type of the bracket: BracketFull, BracketBrace or BracketSub. |
| TopStave | Returns the Staff object at which the bracket starts, relative to the current part. |
| TopStaveNum | Returns the number of the top staff included in the bracket, relative to the current part. |
BracketsAndBraces

Accessed from a Score object. Describes the brackets (which may be brackets, sub-brackets or braces) present in the score.

for each bracket in iterates through all the brackets in the list.

Array access \([int n]\) returns the \(n\)th bracket in the list, or null if the bracket does not exist.

Methods

AddBracket(type, top staff number, bottom staff number)
Creates a bracket of a given type, spanning the range of staves specified between top staff number and bottom staff number inclusive, and returns the new Bracket object. The staff numbers are relative to the current part view. Values for type are BracketFull (0), BracketBrace (1) and BracketSub (2).

ClearAll()
Removes all existing brackets, sub-brackets and braces from the current part, and returns the number of brackets removed.

DeleteNthBracket(n)
Deletes the \(n\)th bracket from the current part, and returns True if successful.

Variables

NumChildren
Returns the number of child brackets, sub-brackets and braces in the list.
Object Reference

Clef

Derived from a BarObject

Methods
None.

Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClefStyle</td>
<td>The name of this clef, which may be different depending on the state of Notes Transposing Score (read only).</td>
</tr>
<tr>
<td>ConcertClefStyleId</td>
<td>The concert pitch identifier of the style of this clef (read only).</td>
</tr>
<tr>
<td>ConcertClefStyle</td>
<td>The concert pitch name of this clef (read only).</td>
</tr>
<tr>
<td>StyleId</td>
<td>The identifier of the style of this clef, which may be different depending on whether or not Notes Transposing Score is switched on. This can be passed to the Bar.AddClef method to create a clef of this style (read only).</td>
</tr>
<tr>
<td>TransposingClefStyle</td>
<td>The transposing score name of this clef (read only).</td>
</tr>
<tr>
<td>TransposingClefStyleId</td>
<td>The transposing score identifier of the style of this clef (read only).</td>
</tr>
</tbody>
</table>
Comment

Derived from a BarObject.

Methods

None; create via Bar object.

Variables

Maximized

Returns True if the comment is maximized, otherwise returns False (read/write).

Text

Returns the text of the comment (read/write).

TextWithFormatting

Returns an array containing the various changes of font or style (if any) within the comment’s text in a new element (read only). For example, “This text is \B\bold\b, and this is \italic\i” would return an array with eight elements containing the following data:

arr[0] = “This text is “
arr[1] = “\B"
arr[2] = “\b”
arr[3] = “\b”
arr[4] = “, and this is “
arr[5] = “\I”
arr[6] = “\italic”
arr[7] = “\i”

TextWithFormattingAsString

The comment’s text including any changes of font or style (read only).

TimeStamp

Returns a DateTime object corresponding to the date the comment was created or last edited (read only).

UserName

Returns the username of the user who created or last edited the comment (read only).
ComponentList

An array that is obtained from Sibelius.HouseStyles or Sibelius.ManuscriptPapers. It can be used in a for each loop or as an array with the [n] operator to access each Component object:

Methods
None.

Variables
NumChildren Number of plug-ins (read only).
Component

This represents a Sibelius "component," namely a house style or a manuscript paper. Examples:

```c
// Create a new score using the first manuscript paper
papers=Sibelius.ManuscriptPapers;
score=Sibelius.New(papers[0]);
// Apply the first house style to the new score
styles=Sibelius.HouseStyles;
score.ApplyStyle(styles[0], "ALLSTYLES");
```

**Methods**

None.

**Variables**

| Name       | The name of the component (read only). |
DateTime

This object returns information about the current date and time.

Methods
None.

Variables
Seconds Returns the number of seconds from the time in a date (read only).
Minutes Returns the number of minutes from the time in a date (read only).
Hours Returns the number of hours from the time in a date (read only).
DayOfMonth returns the nth day on the month, 1-based (read only).
Month returns the nth month of the year, 1-based (read only).
Year returns the year (read only).
NthDayOfWeek returns the nth day of the week, 0-based (read only).
NthDayOfYear returns the nth day of the year, 0-based (read only).
LongDate returns the date in a human-readable format, e.g. 1st May 2008 (read only).
ShortDate returns the date in a human-readable format, e.g. 01/05/2008 (read only).
LongDateAndTime returns the date and time in a human-readable format, e.g. 1st May 2008 14:07 (read only).
ShortDateAndTime returns the date and time in a human-readable format, e.g. 01/05/2008 14:07 (read only).
TimeWithSeconds returns the time in a human-readable format, e.g. 14:07 (read only).
TimeWithoutSeconds returns the time in a human-readable format, e.g. 14:07:23 (read only).
Dictionary

For more details about using dictionaries in ManuScript, see Dictionary on page 22.

To create a dictionary, use the built-in function `CreateDictionary(name1, value1, name2, value2, ... nameN, valueN)`.

This creates a dictionary containing user properties called `name1, name2, nameN` with values `value1, value2, valueN` respectively.

To iterate over dictionaries:

- To iterate over element values in Dictionary objects, use `for each n in Dictionary` or `for each Value n in Dictionary`.
- To iterate over element names in Dictionary objects, use `for each Name n in Dictionary`.
- To iterate over `value.name` pairs in Dictionary objects, use `for each Pair n in Dictionary`; this returns a new Dictionary object: `n.Name` is the element name, `n.Value` is the element value.

Methods

`CallMethod(methodname, param1, param2, ... paramN)`

Calls the specified method `methodname` in the dictionary, passing in any other values that are required for the method as further parameters.

`GetMethodNames()`

Returns a sparse array containing the names of the methods belonging to a dictionary.

`GetPropertyNames()`

Returns a sparse array of the names of all the user properties in the dictionary (same as `_propertyNames`).

`MethodExists(methodname)`

Returns `True` if the specified method `methodname` exists in the dictionary.

`PropertyExists(propertyname)`

Returns `True` if the specified user property `propertyname` exists in the dictionary.

`SetMethod(methodname, Self, method)`

Binds a method to the dictionary. `methodname` is the name by which you want to access the method via the dictionary, `Self` refers to the plug-in in which the method is found, and `method` is the name of the method itself, found elsewhere in the plug-in.

Variables

None.

Converting old-style hash tables to dictionaries

The Dictionary object is, among other things, a replacement for the old Hash object, which was a simple hash table object. You are recommended to use the new Dictionary object instead of the old Hash object in your plug-ins, but if you have an existing plug-in in which old-style hashes are used, you can convert them to Dictionaries as follows:

`Hash.ConvertToDictionary()` returns a new Dictionary object, populated with strings converted from the old-style Hash.
DocumentSetup


When you first access the DocumentSetup object, the units default to millimetres; if you want to use another unit of measurement, set DocumentSetup.Units before you set any of the other values. This will not, however, change the units displayed in Layout > Document Setup; to do that, set DocumentSetup.UnitsInDocumentSetupDialog.

Be careful also that if you set DocumentSetup.PageSize after setting DocumentSetup.PageWidth or DocumentSetup.PageHeight, the page size specified will override any custom height/width you may have just set: so set the page size before you then adjust the width or height of the page.

Methods
None.

Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AboveTopStaveGap</td>
<td>Returns or sets the top staff margin on each page in the units specified by the Units variable (read/write).</td>
</tr>
<tr>
<td>AboveTopStaveGapAfterFirstPage</td>
<td>Returns or sets the top staff margin on pages after the first page in the units specified by the Units variable (read/write). To set this, first set FirstPageHasUniqueVerticalStaveMargins to True.</td>
</tr>
<tr>
<td>BelowBottomStaveGap</td>
<td>Returns or sets the bottom staff margin on each page in the units specified by the Units variable (read/write). To set this, first set FirstPageHasUniqueVerticalStaveMargins to True.</td>
</tr>
<tr>
<td>BelowBottomStaveGapAfterFirstPage</td>
<td>Returns or sets the bottom staff margin on each page after the first page in the units specified by the Units variable (read/write).</td>
</tr>
<tr>
<td>FirstPageHasUniqueVerticalStaveMargins</td>
<td>Returns True if the After first page checkbox is switched on in Document Setup, specifying that the first page of the score has different top and bottom staff margins to subsequent pages; otherwise returns False (read/write).</td>
</tr>
<tr>
<td>Orientation</td>
<td>Returns or sets the current page orientation. Values are OrientationPortrait (0) and OrientationLandscape (1). If you change the orientation, this will swap the PageTopMargin and PageBottomMargin values with the PageLeftMargin and PageRightMargin values, to reflect the change in orientation (read/write).</td>
</tr>
<tr>
<td>PageHeight</td>
<td>Returns or sets the height of a page in the units specified by the Units variable (read/write).</td>
</tr>
<tr>
<td>PageSize</td>
<td>Returns or sets the current page size. Values are listed in PageSize values on page 158. If you attempt to set PageSize to PageSizeCustom, Sibelius will do nothing; to set a custom page size, set PageWidth and PageHeight individually. Setting any default PageSize value will also change the PageWidth and PageHeight values (read/write).</td>
</tr>
<tr>
<td>PageWidth</td>
<td>Returns or sets the width of a page in the units specified by the Units variable (read/write).</td>
</tr>
<tr>
<td>MarginType</td>
<td>Returns or sets the current page margin type. Values are PageMarginsSame (0), PageMarginsMirrored (1), PageMarginsDifferent (2) (read/write).</td>
</tr>
<tr>
<td>PageBottomMargin</td>
<td>Returns or sets the bottom page margin in the units specified by the Units variable (read/write).</td>
</tr>
<tr>
<td>PageLeftMargin</td>
<td>Returns or sets the left page margin in the units specified by the Units variable (read/write).</td>
</tr>
<tr>
<td>PageRightMargin</td>
<td>Returns or sets the right page margin in the units specified by the Units variable (read/write).</td>
</tr>
</tbody>
</table>
### DocumentSetup

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PageTopMargin</td>
<td>Returns or sets the top page margin in the units specified by the <strong>Units</strong> variable (read/write).</td>
</tr>
<tr>
<td>RightPageLeftMargin</td>
<td>Returns or sets the left page margin for right-hand pages in the units specified by the <strong>Units</strong> variable (read/write). Setting this value automatically sets <strong>MarginType</strong> to <strong>PageMarginsDifferent</strong>.</td>
</tr>
<tr>
<td>RightPageRightMargin</td>
<td>Returns or sets the right page margin for right-hand pages in the units specified by the <strong>Units</strong> variable (read/write). Setting this value automatically sets <strong>MarginType</strong> to <strong>PageMarginsDifferent</strong>.</td>
</tr>
<tr>
<td>StaffLeftMarginFullNames</td>
<td>Returns or sets the margin to the left of staves showing full instrument names in the units specified by the <strong>Units</strong> variable (read/write).</td>
</tr>
<tr>
<td>StaffLeftMarginNoNames</td>
<td>Returns or sets the margin to the left of staves showing no instrument names in the units specified by the <strong>Units</strong> variable (read/write).</td>
</tr>
<tr>
<td>StaffLeftMarginShortNames</td>
<td>Returns or sets the margin to the left of staves showing short instrument names in the units specified by the <strong>Units</strong> variable (read/write).</td>
</tr>
<tr>
<td>StaffSize</td>
<td>Returns or sets the staff size in the units specified by the <strong>Units</strong> variable (read/write).</td>
</tr>
<tr>
<td>Units</td>
<td>Returns or sets the units of measurement for all of the relevant variables of the DocumentSetup object. Always returns 0 (millimeters). Values are <strong>DocumentSetupUnitsmm</strong> (0), <strong>DocumentSetupUnitsInches</strong> (1), <strong>DocumentSetupUnitsPoints</strong> (2) (read/write).</td>
</tr>
<tr>
<td>UnitsInDocumentSetupDialog</td>
<td>Returns or sets the units of measurement currently shown in the <strong>Layout</strong> &gt; <strong>Document Setup</strong> dialog. Values are as for <strong>Units</strong>.</td>
</tr>
</tbody>
</table>
DynamicPartCollection


The DynamicPartCollection object always contains the full score as the first entry, whether or not any dynamic parts exist. The DynamicPart objects are returned in the order in which they were created (the last part returned is the most-recently created one). For scores in which dynamic parts were generated automatically, the parts will normally be returned in top to bottom score order.

The edit context for ManuScript is stored in the score itself which means that ManuScript can only ever access one part at a time – the “current” DynamicPart for that Score object. This is irrespective of the number of score windows open for a score, which dynamic parts are open, and even if the user has managed to create two different ManuScript Score objects referring to the same Sibelius score.

It is inadvisable to modify Staves, Bars, or any BarObjects that do not exist on Staves in Score.CurrentDynamicPart. Doing so will create part overrides for part-specific properties of these objects which will be invisible until those Staves are added to the part. DynamicPart.IncludesStaff() can be used to test if a DynamicPart contains a particular Staff object.

Both DynamicPartCollection and DynamicPart refer to an underlying Score and part(s) and will generate errors if the Score and/or part(s) are no longer valid (e.g. if a DynamicPart has been deleted). DynamicParts are never “re-used.” For example, if you delete a DynamicPart and create a new DynamicPart, the old ManuScript DynamicPart object will not refer to the newly-created DynamicPart.

for each variable in iterates through all valid DynamicPart objects for the Score, always starting first with the full score. Adding or deleting parts while iterating will have undefined results, and is not recommended.

Array access [int n] returns the nth part (0 is always the full score), or null if the part does not exist.

Methods

CreateDefaultParts()

Creates the default set of dynamic parts, as created automatically by Sibelius when clicking the New Part button in the Parts window. This method does nothing and returns False if the Score has no staves.

CreatePartFromStaff(staff)

Creates a dynamic part from the specified Staff object, if valid. Returns the new DynamicPart object for success, or null for failure.

DeletePart(dynamic part)

Deletes the specified part, if it’s valid. Returns True for success, False for failure. This method fails is the specified dynamic part is the currently active part for the Score, or is the full score, or refers to a different Score.

Variables

NumChildren Returns the number of DynamicPart objects for the Score returned by iteration (read only).
DynamicPart

Accessed from a DynamicPartCollection object.

for each variable in returns the Staff objects in the dynamic part, in top to bottom order. Warning: this can return a Staff that is not included in Score.CurrentDynamicPart.

Methods

AddStaffToPart(staff)

Adds the specified staff to the bottom of the dynamic part. Returns False for failure. This method will cause an error if it is called on the full score, or if attempting to add a staff that is already present in the part or a staff from a different score.

DeleteStaffFromPart(staff)

Deletes the specified staff from the dynamic part. Returns False for failure. This method will cause an error if called on the full score, or if attempting to delete a staff that is not present in the part, or if deleting the last staff in a part, or attempting to delete a part from a different score.

IncludesStaff(staff)

Returns True if the specified staff is contained in this dynamic part.

Variables

IsFullScore

Returns True if this is the full score (read only).

IsSelectedInPartsWindow

Returns True if the part is selected in the Parts window (read only).

StaveCount

Returns the number of staves in the part (read only).

ParentScore

Returns the Score object containing this dynamic part (read only).
EngravingRules

Accessed via the Score object. Corresponds to selected settings in the House Style > Engraving Rules dialog.

Methods

None.

Variables.

AdjustTranspositionIfKeySigWraps
Returns True if Sibelius will adjust note spelling for transposing instruments in extreme keys, False otherwise; corresponds to the Adjust note spelling in transposing instruments in remote keys option on the Clefs and Key Signatures page (read/write).

BarlineWidth
Returns or sets the width of normal barlines in spaces, from the Barlines page (read/write).

BeamThickness
Returns or sets the thickness of beams in spaces, from the Beams and Stems page (read/write).

CautionaryNaturalsInKeySignatures
Returns True if key changes will show cautionary naturals; False otherwise, from the Clefs and Key Signatures page (read/write).

CueNoteScale
Returns or sets the percentage by which cue-sized notes are scaled relative to normal-sized notes, from the Notes and Tremolos page (read/write).

DashedBarlineWidth
Returns or sets the width of dashed barlines in spaces, from the Barlines page (read/write).

DoubleBarlineSeparation
Returns or sets the distance between the two lines in double barlines in spaces, from the Barlines page (read/write).

DoubleBarlineWidth
Returns or sets the width of double barlines in spaces, from the Barlines page (read/write).

DoubleTremoloStyle
Returns or sets the style used for double tremolos in the score, from the Notes and Tremolos page; values are DoubleTremolosTouchingStems (0), DoubleTremolosBetweenStems (1), DoubleTremolosOuterTremoloTouchingStems (2) (read/write).

ExtraSpacesAboveForSystemObjectPositions
Returns or sets the n extra spaces above for System Object Positions value on the Staves page (read/write).

ExtraSpacesBelowVocalStaves
Returns or sets the n extra spaces below vocal staves (for lyrics) value on the Staves page (read/write).

ExtraSpaceBetweenGroupsOfStaves
Returns or sets the n extra spaces between groups of staves value on the Staves page (read/write).

FinalBarlineSeparation
Returns or sets the distance between the two lines in final barlines in spaces, from the Barlines page (read/write).

FinalBarlineWidth
Returns or sets the width of the thick line of final barlines in spaces, from the Barlines page (read/write).

GraceNoteScale
Returns or sets the percentage by which grace notes are scaled relative to normal notes, from the Notes and Tremolos page (read/write).

InstrumentNamesFirstSystem
Corresponding to the option for instrument names on the first system on the Instruments page.
EngravingRules

InstrumentNamesNewSections
Corresponding to the option for instrument names at the start of new sections on the Instruments page; values are InstrumentNamesFull (0), InstrumentNamesShort (1), InstrumentNamesNone (2) (read/write).

InstrumentNamesSubsequentSystems
Corresponding to the option for instrument names on subsequent systems on the Instruments page; values are InstrumentNamesFull (0), InstrumentNamesShort (1), InstrumentNamesNone (2) (read/write).

JustifyGrandStaveInstruments
Returns True if Justify both staves of grand staff instruments on the Staves page is switched on, otherwise False (read/write).

JustifyMultiStaveInstruments
Returns True if Justify all staves of multi-staff instruments on the Staves page is switched on, otherwise False (read/write).

LegerLineThickness
Returns or sets the thickness of leger lines in spaces, from the Notes and Tremolos page (read/write).

RespellRemoteKeysInTransposingScore
Returns True if Sibelius will choose the equivalent key signature with one fewer flat or sharp for transposing instruments; False otherwise, corresponding to the option Respell remote key signatures in transposing score on the Clefs and Key Signatures page (read/write).

ShowNameOfPrevailingInstrumentChangeAtStartOfSystems
Returns True if Sibelius will update the instrument name at the start of each system to reflect the current instrument change, False otherwise; corresponds to the Change instrument names at start of system after instrument changes option on the Instruments page (read/write).

SlurMiddleThickness
Returns or sets the default thickness of the middle of slurs in spaces, from the Slurs page (read/write).

SlurOutlineWidth
Returns or sets the thickness of slur ends in spaces, from the Slurs page (read/write).

SmallStaffSizeScale
Returns or sets the percentage by which small staves are scaled relative to normal-sized staves, from the Staves page (read/write).

SpacesBetweenStaves
Returns or sets the n spaces between staves value on the Staves page (read/write).

SpacesBetweenSystems
Returns or sets the n spaces between systems value on the Staves page (read/write).

StaffJustificationPercentage
Returns or sets the Justify staves when page is at least n% full value on the Staves page (read/write).

StaffLineWidth
Returns or sets the width of a staff line in spaces, from the Staves page (read/write).

StemThickness
Returns or sets the thickness of stems in spaces, from the Beams and Stems page (read/write).

TieMiddleThickness
Returns or sets the thickness of the middle of ties in spaces, from the Ties 1 page (read/write).

TieOutlineWidth
Returns or sets the thickness of tie ends in spaces, from the Ties 1 page (read/write).
Object Reference

File

Retrievable using for each on a folder.

Methods

Delete()

Deletes a file, returning True if successful.

Rename(newFileName)

Renames a file, returning True if successful.

Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CreationDate</td>
<td>Returns the file's creation date and time as a DateTime object, in local time (read only).</td>
</tr>
<tr>
<td>CreationDateAndTime</td>
<td>A string giving the date and time the file was last modified in GMT (read only).</td>
</tr>
<tr>
<td>ModificationDate</td>
<td>Returns the file's modification date and time as a DateTime object, in local time (read only).</td>
</tr>
<tr>
<td>Name</td>
<td>The complete pathname of the file, no extension (read only).</td>
</tr>
<tr>
<td>NameWithExt</td>
<td>The complete pathname of the file, with extension (read only).</td>
</tr>
<tr>
<td>NameNoPath</td>
<td>Just the name of the file, no extension (read only).</td>
</tr>
<tr>
<td>Path</td>
<td>Returns just the path to the file (read only).</td>
</tr>
<tr>
<td>Type</td>
<td>A string giving the name of the type of the object; File for file objects (read only).</td>
</tr>
</tbody>
</table>
Folder

Retrievable from methods of the Sibelius object.

for each variable in produces the Sibelius files in the folder, as File objects.

for each type variable in produces the files of type type in the folder, where type is a Windows extension. Useful values are SIB (Sibelius files), MID (MIDI files) or OPT (PhotoScore files), because they can all be opened directly by Sibelius. On the Macintosh files of the corresponding Mac OS Type are also returned (so, for example, for each MID f will return all files whose names end in .MID, and all files of type "Midi").

Both these statements return subfolders recursively.

Methods
FileCount (Type)

Returns the number of files of type Type in the folder. As above, useful values are SIB, MID or OPT.

Variables
FileCount
FileCountAllTypes
Name
Type

The number of Sibelius files in the folder (read only).
The number of files of all types in the folder (read only).
The name of the folder (read only).
A string giving the name of the type of the object; Folder for folder objects (read only).
GuitarFrame

Derived from a BarObject. This refers to chord symbols as created by Create Chord Symbol, whether or not they show a guitar chord diagram (guitar frame), but is called GuitarFrame in ManuScript for historical reasons.

Methods

`CopyOutSuffixes()`

Returns an array containing a list of the suffix elements present in the chord. If the chord symbol is an unrecognised chord type, the array returned will be empty. The values that can be returned in the array are as follows:

- halfdim
- dim
- add6/9
- add13
- maj
- maj13
- maj7
- maj9
- maj11
- maj13
- add11
- add5
- add9
- dim
- sus
- sus2
- sus2/4
- sus4
- sus9
- sus11

`GetChromaticPitchesOfChordInClosePosition(consider root)`

Returns an array containing the chromatic pitches of the notes in the chord, assuming a voicing in close position. If consider root is True (it defaults to False), the pitches returned will be offset according to the chromatic value of the root note on which the chord is based.

`GetEndStringForNthBarre(barreNum)`

Returns the string number on which the nth barré ends.

`GetPitchOfNthString(stringNum)`

Returns the pitch of the given (open) string stringNum, as a MIDI pitch.

`GetPositionOfFingerForNthBarre(barreNum)`

Returns the fret position that the nth barré occupies.

`GetPositionOfFingerOnNthString(stringNum)`

Returns the position of the black dot representing the finger position on a given string stringNum, relative to the top of the frame. A return value of 0 means the string is open (i.e. a hollow circle appears at the top of the diagram), and -1 means that the string is not played (i.e. an X appears at the top of the diagram). Used in conjunction with GetPitchOfNthString(), you can calculate the resulting pitch of each string.

`GetStartStringForNthBarre(barreNum)`

Returns the string number from which the nth barré begins.
IsNthStringPartOfBarre(stringNum)

Returns True if the given string is part of a barré.

NthStringHasClosedMarkingAtNut(nth string)

Returns True if there’s an X marking at the top or left of the specified string.

NthStringHasOpenMarkingAtNut(nth string)

Returns True if there’s an O marking at the top or left of the specified string.

Variables

BassAsString

The note name of the chord symbol’s altered bass note (e.g. “F”).

ChordNameAsStyledString

The name of the chord represented by this chord symbol as it appears in the score, e.g. “Cm7” (read only).

ChordNameAsPlainText

The name of the chord represented by this chord symbol as it appears when editing the chord symbol, i.e. in its plain text representation, e.g. “Cmmaj7” (read only).

ChromaticRoot

The chromatic pitch (C = 0, B = 11) of the chord symbol’s root note (read only).

ChromaticBass

The chromatic pitch (C = 0, B = 11) of the chord symbol’s altered bass note (read only).

DiatonicRoot

The diatonic pitch, i.e. the number of the “note name” to which this note corresponds, 7 per octave (0 = C, 1 = D, 2 = E etc.), of the chord symbol’s root note (read only).

DiatonicBass

The diatonic pitch, i.e. the number of the “note name” to which this note corresponds, 7 per octave (0 = C, 1 = D, 2 = E etc.), of the chord symbol’s altered bass note (read only).

Fingerings

The fingerings string for this chord. This is a textual string with as many characters as the guitar frame has strings (i.e. six for standard guitars). Each character corresponds to a guitar string. Use “-” to denote that a string has no fingering.

FrameIsVisible

True if the chord symbol is currently showing a guitar chord diagram (read only).

Horizontal

True if the guitar chord diagram is horizontally orientated, False if it is vertically orientated (read/write).

LowestVisibleFret

The number of the top fret shown in the guitar chord diagram; setting the special value -1 resets the lowest visible fret to the default for that chord diagram (read/write).

NumBarresInChord

The number of unique barrés in the guitar chord diagram (read only).

NumberOfFrets

The number of frets in the guitar chord diagram, i.e. the number of horizontal lines; setting the special value -1 resets the number of frets to the default for that chord diagram (read/write).

NumberOfStrings

The number of strings in the guitar chord diagram, i.e. the number of vertical lines (read only).

NumPitchesInClosePosition

The number of unique pitches in the chord, assuming a voicing in close position with no duplicates.

Recognized

Returns True if the chord symbol is a specific recognized chord type, and False otherwise, i.e. if the chord symbol is shown in red in the score because Sibelius is unable to parse the user’s input (read only).

RootAsString

The note name of the chord symbol’s root (e.g. “C#”).

ScaleFactor

The scale factor of the guitar chord diagram (as adjustable via the Scale parameter on the General panel of Properties), expressed as a percentage (read/write).

ShowFingerings

Set to True if the fingerings string should be displayed, False otherwise (read only).
Object Reference

**SuffixText**
The suffix part of the chord symbol as it appears in the score, or an empty string if the chord isn't recognised (read only).

**TextIsVisible**
True if the chord symbol is currently showing a text chord symbol (read only).

**TransposingChromaticRoot**
Returns the chromatic pitch of the root note for the specified chord symbol as if the score is shown at transposed pitch, but regardless of whether or not Notes> Transposing Score is switched on.

**TransposingChromaticBass**
Returns the chromatic pitch of the altered bass note for the specified chord symbol, if present, as if the score is shown at transposed pitch, but regardless of whether or not Notes> Transposing Score is switched on.

**TransposingDiatonicRoot**
Returns the diatonic pitch of the root note for the specified chord symbol as if the score is shown at transposed pitch, but regardless of whether or not Notes> Transposing Score is switched on.

**TransposingDiatonicBass**
Returns the diatonic pitch of the altered bass note for the specified chord symbol, if present, as if the score is shown at transposed pitch, but regardless of whether or not Notes> Transposing Score is switched on.

**TransposingRootAsString**
Returns a string representing the pitch of the root note for the specified chord symbol as if the score is shown at transposed pitch, but regardless of whether or not Notes> Transposing Score is switched on.

**TransposingBassAsString**
Returns a string representing the pitch of the altered bass note for the specified chord symbol, if present, as if the score is shown at transposed pitch, but regardless of whether or not Notes> Transposing Score is switched on.

**VisibleComponents**
The visible parts of the chord symbol, i.e. whether it displays a text chord symbol only (TextOnly), a guitar chord diagram only (FrameOnly), both a text chord symbol and a guitar chord diagram (FrameAndText), or whether or not the chord symbol shows a guitar chord diagram based on the type of instrument to which it is attached (InstrumentDependent) (read/write).
GuitarScaleDiagram

Derived from a BarObject. This refers to guitar scale diagrams as created by Create > Guitar Scale Diagram.

Methods

GetDotFingeringsOnNthString(nth string)

Returns an array of strings containing the text that has been entered on the dots on a given string.

GetDotSymbolsOnNthString(nth string)

Returns an array of values describing the appearance of each of the dots on a given string. The possible values are DotStyleCircle, DotStyleFilledCircle, DotStyleSquare, DotStyleFilledSquare, DotStyleDiamond, and DotStyleFilledDiamond.

GetPitchesOfDotsOnNthString(nth string)

Returns an array containing the pitches of all the dots on a given string, in ascending order of pitch.

GetPitchOfNthString(stringNum)

Returns the pitch of the given (open) string stringNum, as a MIDI pitch.

Variables

Fingerings

The fingerings string for this scale diagram. This is a textual string with as many characters as the scale diagram has strings (i.e. six for standard guitars). Each character corresponds to a guitar string. Use - to denote that a string has no fingering.

Horizontal

True if the guitar scale diagram is horizontally orientated, False if it is vertically orientated (read/write).

LowestVisibleFret

The number of the top fret shown in the guitar scale diagram; setting the special value -1 resets the lowest visible fret to the default for that scale diagram (read/write).

NumberOfFrets

The number of frets in the guitar scale diagram, i.e. the number of horizontal lines; setting the special value -1 resets the number of frets to the default for that scale diagram (read/write).

NumberOfStrings

The number of strings in the guitar scale diagram, i.e. the number of vertical lines (read only).

Root

Returns the chromatic pitch (C = 0) of the scale's root note (read only).

ScaleFactor

The scale factor of the guitar scale diagram (as adjustable via the Scale parameter on the General panel of Properties), expressed as a percentage (read/write).

ScaleType

Returns the type of the guitar scale diagram, as specified in the list of GuitarScaleDiagram type values on page 157 (read only).

ShowFingerings

Set to True if the fingerings string should be displayed, False otherwise (read only).
Object Reference

HitPointList

Retrievable as the HitPoints variable of a score. It can be used in a for each loop or as an array with the [n] operator – this
gives access to a HitPoint object. The HitPoint objects are stored in time order, so be careful if you remove or modify the time of
the objects inside a loop. If you want to change the times of all the hit points by the same value then use the ShiftTimes
function.

Methods

Clear()

Removes all hit points from the score.

CreateHitPoint (timeMs, label)

Creates a hit point in the score at the given time (specified in milliseconds) with a specified string label. Returns the index in
the HitPointList at which the new hit point was created.

Remove (index)

Removes the given hit point number.

ShiftTimes (timeMs)

Adds the given time (in milliseconds) onto all the hit points. If the time is negative then this is subtracted from all the hit
points.

Variables

NumChildren Number of hit points (read only).
HitPoint

Individual element of the HitPointList object.

Methods
None.

Variables
- **Bar**
  The bar in which this hit point occurs (read only).
- **Label**
  The name of the hit point (read/write).
- **Position**
  The position within the bar at which this hit point occurs (read only).
- **Time**
  The time of the hit point in milliseconds. Note that changing this value may change the position of the hit point in the HitPointList (read/write).
Object Reference

InstrumentChange

Derived from a Bar object. Provides information about any instrument changes that may exist in the score.

Methods
None.

Variables

- **StyleId**
  Returns the style ID of the new instrument; see Instrument types on page 139 (read only).

- **TextLabel**
  Returns the text that appears above the staff containing the instrument change in the score (read only).
**InstrumentTypeList**

Contains a list of `InstrumentType` objects common to a given score.

*for each type variable in* returns each instrument type in the list, in alphabetical order by the instrument type's style ID.

*Array access [int n]* returns the nth instrument type, in the same order as using a *for each* iterator, or null if the instrument type does not exist.

**Methods**

None.

**Variables**

- `NumChildren`  
  Returns the number of unique instrument types in the list (read only).
Object Reference

InstrumentType

Provides information about an individual instrument type.

Methods

Clone() Makes an exact copy of an existing instrument type.

PitchOfNthString(string num)
Returns the pitch of a given string in a tablature staff, with string number 0 being the lowest string on the instrument.

Variables

Balance Returns the instrument’s default balance, in the range 0–100 (read only).

Category Returns an index representing the category of the staff type belonging to this instrument type; 0 = pitched; 1 = percussion; 2 = tablature (read only).

ChromaticTransposition Returns the number of half-steps (semitones) describing the transposition of transposing instruments; e.g. for B♭ Clarinet, this returns -2 (read/write).

ChromaticTranspositionInScore Returns the number of half-steps (semitones) describing the transposition of transposing instruments in a score shown at concert pitch. Typically this is only used by instruments that transpose by octaves, so this will return e.g. 12 for piccolo or -12 for guitars (read only).

ComfortableRangeHigh Returns the highest comfortable note (MIDI pitch) of the instrument (read only).

ComfortableRangeLow Returns the lowest comfortable note (MIDI pitch) of the instrument (read only).

ConcertClefStyleId Returns the style ID of the normal clef style of the instrument (read only).

DefaultSoundId Returns the default sound ID used by the instrument (read only).

DiatonicTransposition Returns the number of diatonic steps describing the transposition of transposing instruments; e.g. for B♭ Clarinet, this returns -1 (read/write).

DiatonicTranspositionInScore Returns the number of diatonic steps describing the transposition of transposing instruments in a score shown at concert pitch (read only).

DialogName Returns the name of the instrument as displayed in the Create > Instruments dialog in Sibelius (read/write).

FullName Returns the name of the instrument as visible on systems showing full instrument names (read only).

HasBracket Returns True if the instrument has a bracket (read only).

HasKeySignatureOrTuning Returns True if the instrument type has the Key signature / Tuning checkbox switched on in the Edit Staff Type dialog.

InstrumentTypeForChordDiagrams Returns the style ID of the tab instrument type that determines the tuning used for chord diagrams shown for this instrument, i.e. corresponding to the Tab instrument to use for string tunings in the New/Edit Instrument dialogs.

IsVocal Returns True if the instrument type used has the Vocal staff option switched on, meaning that e.g. the default positions of dynamics should be above the staff rather than below (read only).

NumStaveLines Returns the number of staff lines in the staff (read only).

NumStrings Returns the number of strings in a tablature staff (read only).
<table>
<thead>
<tr>
<th><strong>OtherClefStyleId</strong></th>
<th>Returns the style ID of the clef style of the second staff of grand staff instruments, e.g. piano (read only).</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pan</strong></td>
<td>Returns the instrument's default pan setting, in the range –127 (hard left) to 127 (hard right) (read only).</td>
</tr>
<tr>
<td><strong>ProfessionalRangeHigh</strong></td>
<td>Returns the highest playable note (MIDI pitch) of the instrument for a professional player (read only).</td>
</tr>
<tr>
<td><strong>ProfessionalRangeLow</strong></td>
<td>Returns the lowest playable note (MIDI pitch) of the instrument for a professional player (read only).</td>
</tr>
<tr>
<td><strong>ShortName</strong></td>
<td>Returns the name of the instrument as visible on systems showing short instrument names (read only).</td>
</tr>
<tr>
<td><strong>StyleId</strong></td>
<td>Returns the style ID of the instrument; see <strong>Global constants</strong> on page 135 (read only).</td>
</tr>
<tr>
<td><strong>TransposingClefStyleId</strong></td>
<td>Returns the style ID of the clef to be used when <strong>Notes &gt; Transposing Score</strong> is switched on (read only).</td>
</tr>
</tbody>
</table>
KeySignature

Derived from a BarObject.

Methods
None

Variables

AsText  The name of the key signature as a string (read only).
IsOneStaffOnly  True if this key signature belongs to one staff only (read only).
Major  True if this key signature is a major key (read only).
Sharps  The number of sharps (positive) or flats (negative) in this key signature (read only).
Anything you can create from the Create Line dialog is a line object, eg. CrescendoLine, DiminuendoLine, etc. These objects are derived from a BarObject.

**Methods**
None.

**Variables**

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Duration</strong></td>
<td>The total duration of the line, in 1/256th quarters (read/write).</td>
</tr>
<tr>
<td><strong>EndBarNumber</strong></td>
<td>The bar number in which the line ends (read only).</td>
</tr>
<tr>
<td><strong>EndPosition</strong></td>
<td>The position within the final bar at which the line ends (read only).</td>
</tr>
<tr>
<td><strong>RhDx</strong></td>
<td>The horizontal graphic offset of the right-hand side of the line, in units of 1/32 spaces (read/write).</td>
</tr>
<tr>
<td><strong>RhDy</strong></td>
<td>The vertical graphic offset of the right-hand side of the line from the centre staff line, in units of 1/32 spaces, positive going upwards (read/write).</td>
</tr>
<tr>
<td><strong>StyleId</strong></td>
<td>The identifier of the line style associated with this line (read only).</td>
</tr>
<tr>
<td><strong>StyleAsText</strong></td>
<td>The name of the line style associated with this line (read only).</td>
</tr>
</tbody>
</table>
Object Reference

LyricItem

Derived from a BarObject

Methods
None.

Variables

- **Duration**
  The total duration of the lyric line, in 1/256th quarters (see Line on page 85) (read/write).

- **NumNotes**
  Gives the number of notes occupied by this lyric item (read/write). Note that changing this value will not automatically change the length of the lyric line; you also need to set the lyric line's Duration variable to the correct length.

- **StyleAsText**
  The text style name (read/write).

- **StyleId**
  The identifier of the text style of this lyric (read/write).

- **SyllableType**
  An integer indicating whether the lyric is the end of a word (EndOfWord) or the start or middle of one (MiddleOfWord) (read/write). This affects how the lyric is justified, and the appearance of hyphens that follow it. EndOfWord and MiddleOfWord are global constants; see SyllableTypes for LyricItems on page 154.

- **Text**
  The text as a string (read/write).
NoteRest

Derived from a BarObject. A NoteRest contains Note objects, stored in order of increasing diatonic pitch.

for each variable in returns the notes in the NoteRest.

Methods

AddAcciaccaturaBefore (sounding pitch, [duration[, tied [, voice [, diatonic pitch[, string number[, force stem dir]]]]]])

Adds a grace note with a slash on its stem (acciaccatura) before a given NoteRest. The duration should be specified as normal, for example, 128 would create a grace note with one beam/flag. The optional tied parameter should be True if you want the note to be tied. Voice 1 is assumed unless the optional voice parameter (with a value of 1, 2, 3 or 4) is specified. If force stem dir is set to True (the default), stems of graces notes in voices 1 and 3 will always point upwards, and stems of notes in voices 2 and 4, downwards. You can also set the diatonic pitch, i.e. the number of the "note name" to which this note corresponds, 7 per octave (35 = middle C, 36 = D, 37 = E and so on). If a diatonic pitch of zero is given then a suitable diatonic pitch will be calculated from the MIDI pitch. The optional string number parameter gives a string number for this note, which is only meaningful if the note is on a tablature stave. If this parameter is not supplied then a default string number is calculated based on the current tablature stave type and the guitar tab fingering options (specified on the Note Input page of File > Preferences). Returns the Note object created (to get the NoteRest containing the note, use Note.Rest.ParentNoteRest).

Note that adding a grace note before a NoteRest will always create an additional grace note, just to the left of the note/rest to which it is attached. If you wish to create grace notes with more than one pitch, you should call AddNote on the object returned.

AddAppoggiaturaBefore (sounding pitch, [duration[, tied [, voice [, diatonic pitch[, string number[, force stem dir]]]]]])

Identical to AddAcciaccaturaBefore, only no slash is added to the note's stem.

AddNote (pitch[, tied[, diatonic pitch[, string number]]])

Adds a note with the given MIDI pitch (60 = middle C), e.g. to create a chord. The optional second parameter specifies whether or not this note is tied (True or False). The optional third parameter gives a diatonic pitch, i.e. the number of the 'note name' to which this note corresponds, 7 per octave (35 = middle C, 36 = D, 37 = E etc.). If this parameter is 0 then a default diatonic pitch will be calculated from the MIDI pitch. The optional fourth parameter gives a string number for this note, which is only meaningful if the note is on a tablature stave. If this parameter is not supplied then a default string number is calculated based on the current tablature stave type and the guitar tab fingering options (specified on the Notes page of File > Preferences). Returns the Note object created.

Delete ()

Deletes all the notes in the NoteRest, converting the entire chord into a rest of similar duration.

FlipStem ()

Flips the stem of this NoteRest – this acts as a toggle.

GetArticulation (articulation number)

Returns True or False depending on whether the given articulation is currently set on this note. The valid articulation numbers are defined in Articulations on page 153.

NoteRest [array element]

Returns the nth note in the chord, in order of increasing diatonic pitch (counting from 0). For example, NoteRest[0] returns the lowest note (in terms of diatonic pitch – see AddNote below).

RemoveNote (note)

Removes the specified Note object.

SetArticulation (articulation number, set)

If set is True, turns on the given articulation; otherwise turns it off. The valid articulation numbers are defined in Articulations on page 153.
Object Reference

**Transpose**(degree, interval type[, keep double accs])

Transposes the entire NoteRest up or down by a specified degree and interval type. To transpose up, use positive values for degree; to transpose down, use negative values. Note that degrees are 0-based, so 0 is equal to a unison, 1 to a second and so on. For descriptions of the various available interval types, see Global constants on page 135. By default, Sibelius will transpose using double sharps and flats where necessary, but this behavior may be suppressed by setting the keep double accs flag to False.

For help in calculating the interval and degree required for a particular transposition, see the documentation for the Sibelius.CalculateInterval and Sibelius.CalculateDegree methods.

**Variables**

- **ArpeggioDx**
  
  The horizontal offset of the arpeggio line on the NoteRest (read/write), in units of 1/32nd of a space (the distance between two adjacent staff lines).

- **ArpeggioType**
  
  The type of note-attached arpeggio line present on the NoteRest. Values are ArpeggioType-None, ArpeggioTypeNormal, ArpeggioTypeUp, ArpeggioTypeDown (read/write).

- **ArpeggioTopDy**
  
  The vertical offset of the top of the note-attached arpeggio line on the NoteRest (read/write), in units of 1/32nd of a space.

- **ArpeggioBottomDy**
  
  The vertical offset of the bottom of the note-attached arpeggio line on the NoteRest (read/write), in units of 1/32nd of a space.

- **ArpeggioHidden**
  
  Returns True if the note-attached arpeggio line on the NoteRest is hidden (read/write).

- **Articulations**
  
  Enables you to copy a set of articulations from one NoteRest to another (read/write), e.g:
  
  destNr.Articulations = sourceNr.Articulations;

- **Beam**
  
  Takes values StartBeam, ContinueBeam, NoBeam and SingleBeam. (see Global constants on page 135 for details). These correspond to the keys 7, 8, */ (/ on Mac) and / (* on Mac) on the third (F9) Keypad layout.

- **DoubleTremolos**
  
  Gives the number of double tremolo strokes starting at this note, in the range 0–7. Means nothing for rests. To create a double tremolo between two successive notes, ensure they have the same duration and set the DoubleTremolos of the first one (read/write).

- **Duration**
  
  The duration of the note rest (read only).

- **FallDx**
  
  The horizontal offset of a fall, if present on the NoteRest (read/write), in units of 1/32nd of a space.

- **FallType**
  
  The type of note-attached fall present on the NoteRest. Values are FallTypeNone, FallTypeNormal and FallTypeDoit (read/write)

- **FeatheredBeamType**
  
  Returns one of three values, based on whether a note is set to produce a feathered beam. Values are FeatheredBeamNone (0), FeatheredBeamAccel (1) and FeatheredBeamRit (2) (read/write).

- **GraceNote**
  
  True if it’s a grace note (read only).

- **HasStemlet**
  
  Returns True if the note is showing a stemlet, according either to the state of the Use stemlets on beamed rests option on the Beams and Stems page of Engraving Rules or the stemlet button on the Keypad (read only).

- **Highest**
  
  The highest Note object in the chord (read only).

- **IsAcciaccatura**
  
  True if it’s an acciaccatura, i.e. a grace note with a slash through its stem (read only).

- **IsAppoggiatura**
  
  True if it’s an appoggiatura, i.e. a grace note without a slash through its stem (read only).

- **Lowest**
  
  The lowest Note object in the chord (read only).

- **NoteCount**
  
  The number of notes in the chord (read only).
**NoteRest**

**ParentTupletIfAny**
If the NoteRest intersects a tuplet, the innermost Tuplet object at that point in the score is returned. Otherwise, `null` is returned (read only).

**PositionInTuplet**
Returns the position of the NoteRest relative to the duration and scale-factor of its parent tuplet. If the NoteRest does not intersect a tuplet, its position within the parent Bar is returned as usual (read only).

**RestPosition**
The vertical position of a rest (read/write).

**ScoopDx**
The horizontal offset of a scoop or plop, if present on the NoteRest (read/write), in units of 1/32nd of a space.

**ScoopType**
The type of note-attached scoop present on the NoteRest. Values are `ScoopTypeNone`, `ScoopTypeNormal`, `ScoopTypePlop` (read/write).

**StemFlipped**
`True` if the stem is flipped (read only).

**StemletType**
Provides information about whether the NoteRest is set to display a stemlet using the options on the Keypad. Returns either `StemletCustomOff` (in which case the NoteRest definitely does not show a stemlet), `StemletCustomOn` (in which case the NoteRest definitely does show a stemlet), or `StemletUseDefault` (in which case you should use the read-only variable `HasStemlet` to determine whether the NoteRest currently shows a stemlet) (read/write).

**Stemweight**
Returns the stemweight of a note, taking beams into account (read only). For an unbeamed note, this is the sum of the stave positions of all the notes in the NoteRest, where the stave position of the middle line is zero and the position increases as you move up the stave and decreases as you move downwards. For a beamed note, it is the sum of all the stemweights of the NoteRests under the beam (treated as though they were unbeamed).

There are some special cases. If a note has its stem direction forced due to voicing, then the stemweight will be one of the global constants `StemweightUp` or `StemweightDown`. If a note has its stem direction forced due to the “flip” flag being set, the stemweight will be either `StemweightFlipUp` or `StemweightFlipDown`. Finally, cross-stave notes have stemweight equal to `StemweightCross`.

If the stemweight is less than zero, the stem will point up, otherwise it will point down.

**SingleTremolos**
Gives the number of tremolo strokes on the stem of this note, in the range -1 (for “z on stem”) to 7. Means nothing for rests (read/write).
Note

Methods

Delete()

Removes a single note from a chord.

Transpose (degree, interval type[, keep double accs])

Transposes and returns a single Note object up or down by a specified degree and interval type*. To transpose up, use positive values for degree; to transpose down, use negative values. Note that degrees are 0-based, so 0 is equal to a unison, 1 to a second and so on. For descriptions of the various available interval types, see Global constants on page 135. By default, Sibelius will transpose using double sharps and flats where necessary, but this behavior may be suppressed by setting the keep double accs flag to False. For help in calculating the interval and degree required for a particular transposition, see the documentation for the Sibelius.CalculateInterval and Sibelius.CalculateDegree methods.

* N.B.: Individual note objects cannot be transposed diatonically.

Variables

Accidental

The accidental, for which global constants such as Sharp, Flat and so on are defined; see Global constants on page 135 (read only).

AccidentalStyle

The style of the accidental (read/write). This can be any of following four global constants: NormalAcc, HiddenAcc, CautionaryAcc (which forces an accidental to appear always) and BracketedAcc (which forces the accidental to be drawn inside brackets).

Bracketed

The bracketed state of the note, as shown on the F9 layout of the Keypad (read/write).

DiatonicPitch

The diatonic pitch of the note, i.e. the number of the “note name” to which this note corresponds, 7 per octave (35 = middle C, 36 = D, 37 = E and so on) (read only).

Name

The pitch of the note as a string (read only).

NoteStyle

The index of the notehead style of this Note (read/write). The styles correspond to those accessible from the Notes panel of the Properties window in Sibelius; see Note Style names on page 153 for a complete list of the defined NoteStyles.

NoteStyleName

The name of the notehead style of this Note (read/write). If an attempt is made to apply a non-existent style name, the note in question will retain its current notehead.

OriginalDeltaSr

The Live start position of this notehead (in 1/256th quarters), as shown in the Playback panel of Properties (read/write). This value can be positive or negative, indicating that the note is moved forwards or backwards.

OriginalDuration

The Live duration of this notehead (in 1/256th quarters), as shown in the Playback panel of Properties (read/write). Note that the word “original” refers to the fact that this data is preserved from the original performance if the score was imported from a MIDI file or input via Flexi-time. For further details on this value, and the ones following below, read the Live Playback section in Sibelius Reference.

OriginalVelocity

The Live velocity of this notehead (in MIDI volume units, 0-127), as shown in the Playback panel of Properties (read/write). Note that the word “original” refers to the fact that this data is preserved from the original performance if the score was imported from a MIDI file or input via Flexi-time. For further details on this value, and the ones following below, read the Live Playback section in Sibelius Reference.

ParentNoteRest

The NoteRest object that holds this note (read only).

Pitch

The MIDI pitch of the note, in semitones, 60 = middle C (read only).

Slide

Is True if the note has a slide, False otherwise (read/write).
**StringNum**  
The string number of this note, only defined if the note is on a tablature stave. If no string is specified, reading this value will give -1. Strings are numbered starting at 0 for the bottom string and increasing upwards (read only).

**Tied**  
Is True if the note is tied to the following note (read/write).

**WrittenAccidental**  
The accidental, taking transposition into account (read only).

**WrittenDiatonicPitch**  
The written diatonic pitch of the note, taking transposition into account if `Score.TransposingScore` is True (35 = middle C) (read only).

**WrittenName**  
The written pitch of the note as a string (taking transposition into account) (read only).

**WrittenPitch**  
The written MIDI pitch of the note, taking transposition into account if `Score.TransposingScore` is True (60 = middle C) (read only).

**UseOriginalDeltaSrForPlayback**  
Is True if the Live start position of this Note should be used for Live Playback. Corresponds to the Live start position checkbox in the Playback panel of the Properties window.

**UseOriginalDurationForPlayback**  
Is True if the Live duration of this Note should be used for Live Playback. Corresponds to the Live duration checkbox in the Playback panel of the Properties window.

**UseOriginalVelocityForPlayback**  
Is True if the Live velocity of this Note should be used for Live Playback. Corresponds to the Live velocity checkbox in the Playback panel of the Properties window.
Object Reference

**PageNumberChange**

Provides access to get and set the attributes of a page number change at the end of a bar or on a blank page.

**Methods**

**SetFormatChangeOnly** *(format change only)*

If *format change only* is **True**, this has the same effect as switching off the **New page number** check box on the **Page Number Change** dialog in Sibelius. The page numbering will therefore continue counting consecutively, but it's possible to (for example) hide a group of page numbers and restore visibility at a later point on the score without having to keep track of the previous page numbers.

**SetHideOrShow** *(page number visibility)*

Takes one of the three **Page number visibility** global constants to determine the visibility of the initial page number change and its subsequent pages; see **Global constants** on page 135.

**SetPageNumber** *(page number)*

Takes an integral number specifying the new number you wish to assign to the page.

**SetPageNumberFormat** *(format)*

Takes one of the four **Page number format** global constants to change the format used to display the page number change; see **Global constants** on page 135.

**Variables**

**HideOrShow**

Returns one of the three **Page number visibility** global constants; see **Global constants** on page 135 (read only).

**PageNumber**

Returns the page number expressed as an integer. For example, page **x** when using Roman numerals would be **10**, or **24** with alphabets (read only).

**PageNumberAsString**

Returns the page number change as visible on the corresponding page in Sibelius (read only).

**PageNumberBlankPageOffset**

Returns the blank page offset of the page number change, or **0** if there are no blank pages following the bar containing the page number change (read only).

**PageNumberFormat**

Returns one of four **Page number format** global constants describing the format of the page number change; see **Global constants** on page 135 (read only).
PluginList

An array that is obtained from `Sibelius.Plugins`. It can be used in a `for each` loop or as an array with the `[n]` operator to access each Plugin object.

**Methods**

- `Contains(pluginName)`
  
  Returns `True` if a plug-in with the given name is installed. This can be used to query whether a plugin is installed before you try to call it.

**Variables**

- `NumChildren` Number of plug-ins (read only).
**Plugin**

This represents an installed plugin. Typical usage:

```plaintext
for each p in Sibelius.Plugins

    trace("Plugin: " & p.Name);

```

**Methods**

The following methods are intended to allow you to check the existence of specific methods, data and dialogs in plug-ins, which allows you to check in advance that e.g. calling a method in another plug-in will succeed, and fail gracefully if the method is not found:

- **MethodExists(method)**
  Returns True if the specified method exists in the current Plugin object.

- **DataExists(data)**
  Returns True if the specified data exists in the current Plugin object.

- **DialogExists(dialog)**
  Returns True if the specified dialog exists in the current Plugin object.

**Variables**

- **File**
  The File object corresponding to the file that the plug-in was loaded from (read only).

- **Name**
  The name of the plug-in (read only).
RehearsalMark

Derived from a BarObject and found in the system staff only. RehearsalMarks have an internal numbering and a visible text representation, both of which can be read from ManuScript.

**Methods**
None.

**Variables**

- **Mark**
  The internal number of this rehearsal mark. By default rehearsal marks are consecutive (with the first one numbered zero), but the user can also create marks with specific numbers.

- **MarkAsText**
  The textual representation of this rehearsal mark as drawn in the score. This is determined by the House Style > Engraving Rules options, and can take various forms (numerical or alphabetical).
Object Reference

Score

A Score contains one SystemStaff and one or more Staff objects.

- For each variable in returns each staff in the score or the current dynamic part in turn (not the system staff).
- For each type variable in returns the objects in the score in chronological order, from the top staff to the bottom staff (for simultaneous objects) and then from left to right (again, not including the system staff).

Methods

AddBars(n)

Adds n bars to the end of the score.

ApplyStyle(style file,"style", ["style"])

Imports named styles from the given house style file (.lib) into the score. The style file parameter can either be a full path to the file, or just the name of one of the styles that appears in the House Style > Import House Style dialog. You can import as many “style” elements as you like in the same method. Style names are as follows:

- HOUSE, TEXT, SYMBOLS, LINES, NOTEHEADS, CLEFS, DICTIONARY, SPACINGRULE, DEFAULTPARTAPPEARANCE, INSTRUMENTSANDENSEMBLES, MAGNETICLAYOUTOPTIONS or ALLSTYLES.

For instance:

    score2.ApplyStyle("C:\NewStyle.lib", "HOUSE", "TEXT");

Note that the constant HOUSE refers, for historical reasons, only to those options in the House Style > Engraving Rules and Layout > Document Setup dialogs, not the entire house style. To import the entire House Style, use the ALLSTYLES constant.

ClefStyleId(clef style name)

Returns the identifier of the clef style with the given name, or the empty string if there is no such clef style.

CreateInstrument(style ID[, change names, ["full name", "short name"])]

Creates a new instrument, given the style ID of the instrument type required (see Instrument types on page 139). If you want to supply the instrument names to be used in the score, set the optional change names parameter to True, then supply strings for the full name and short name. Returns True if the instrument was created successfully and False if the instrument type could not be found.

CreateInstrumentAtBottom(style ID[, change names, ["full name", "short name"])]

Behaves the same way as CreateInstrument, only the new instrument is always created below all other instruments that currently exist in the score. This can be useful when programatically copying a list of staves/instruments from one score to another, as you can guarantee the ordering of the staves will be the same in both scores.

CreateInstrumentAtBottomReturnStave(style ID[, change names, ["full name", "short name"])]

As above, but returns the Staff object created, or null if unsuccessful.

CreateInstrumentAtTop(style ID[, change names, ["full name", "short name"])]

Behaves in exactly the same way as CreateInstrumentAtBottom, only the new instrument is always created above all other instruments that currently exist in the score.

CreateInstrumentAtTopReturnStave(style ID[, change names, ["full name", "short name"])]

As above, but returns the Staff object created, or null if unsuccessful.

CreateInstrumentReturnStave(style ID[, change names, ["full name", "short name"])]

Like CreateInstrument, but returns the Staff object created, or null if unsuccessful. Note that if the instrument being created contains more than one staff (e.g. piano or harp), the top stave of the instrument in question will be returned.
Exports one dynamic part, a selection of dynamic parts, or all dynamic parts in PDF format, either concatenated into a single file, or as separate files. The filename parameter should be a complete path. It may contain the following tokens, which Sibelius will expand automatically to generate a complete filename:

- %f = Score filename
- %t = Score title (as specified in the Title field in File > Info)
- %p = Part name (as specified in the Part name field in File > Info)
- %n = Part number
- %o = Total number of parts
- %d = Date (format YYYY-MM-DD)
- %h = Time (format HHMM)

The Boolean parameter single file specifies whether the chosen parts should be extracted into separate PDF files or concatenated into a single PDF file. This parameter defaults to True if not specified.

To specify which parts to export, create a sparse array of part IDs, and pass this in as the third parameter, part IDs. For example:

```csharp
s = Sibelius.ActiveScore;
partsToExport = CreateSparseArray();
parts = s.DynamicParts;
firstNPartsToExport = 2;
i = 0;
for each part in parts {
    if (i <= firstNPartsToExport) { // <= because the first "part" in the
        // DynamicPartsCollection is the full score.
        partsToExport.Push(part);
    }
    i = i + 1;
}
s.ExportPartsAsPDF("c:\%f - %p.pdf", true, partsToExport);
```

To export all parts, pass in 0 instead of a sparse array.

The final optional Boolean parameter, include score, defaults to False. If set to True, the full score will also be exported along with the parts.

Exports the full score as a PDF, with the specified filename, which should be a complete path. The filename parameter may use the same tokens as the ExportPartsAsPDF() method – see above.

Extracts parts from the score. The first optional Boolean parameter can be False, in which case the parts are extracted without showing an options dialog. The second optional parameter specifies a folder into which to extract the parts (must end with a trailing folder separator). The third optional Boolean parameter, which defaults to True, specifies whether the extracted parts should be opened immediately, or simply saved.

Does the same as selecting the whole score and choosing Layout > Magnetic Layout > Freeze Positions, i.e. explicitly sets the Dx/Dy of every object to the position produced by Magnetic Layout, then disables Magnetic Layout for each object.

Returns the time of a given location in the score in milliseconds.

Returns the score’s VersionHistory object (see VersionHistory on page 131).

Inserts n bars before bar number barNum. If no length has been specified, the bar will be created with the correct length according to the current time signature. However, irregular bars may also be created by specifying a value for length.
Object Reference

**InternalPageNumToExternalPageNum** *(pagenum)*

Returns a string containing the external page number of the given internal page number *pagenum*.

**LineStyleId** *(line style name)*

Returns the identifier of the line style with the given name, or the empty string if there is no such line style.

**NoteStyleIndex** *(notehead style name)*

Returns the index of the note style with the given name, or -1 if there is no such note style.

**NthStaff** *(staff index from 1)*

Returns the nth staff of the score or the current dynamic part.

**OptimizeStaffSpacing** *(from staff number[, to staff number[, from bar[, to bar]]])*  

Does the equivalent of **Layout > Optimize Staff Spacing** for the given range of staves or a whole score. *from staff number* must be specified; if *to staff number* is not specified, Sibelius will optimize the distances between *from staff number* and the bottom staff in the score; if *from bar* is not specified, Sibelius sets it to 1; if *to bar* is not specified, Sibelius sets it to the last bar of the score.

**PlayLiveTempo** *(play)*

Switches **Play > Live Tempo** on or off; set *play* to **True** to switch it on, or **False** to switch it off.

**RemoveAllHighlights()**

Removes all highlights in this score.

**RemoveVideo()**

Removes an attached video from the score.

**RenameTextStyle** *("old name", "new name")*

Renames a text style to a new name.

**Save** *(filename)*

Saves the score, overwriting any previous file with the same name.

**SaveAs** *(filename, type[, use_defaults, foldername])*

Saves the score in a specified format, overwriting any previous file with the same name. The optional argument *use_defaults* only applies to graphics files, and specifies whether or not the default settings are to be used. When set to **False**, the **Export Graphics** dialog will appear and allow the user to make any necessary adjustments. The optional *foldername* specifies the folder in which the file is to be saved, and will create the specified folder if it does not exist. The foldername parameter must not end with a path separator (i.e. "\" on Windows).

The possible values for type are:

- **SIBL** Sibelius format (current version)
- **EMF** EMF
- **BMP** Windows bitmap
- **PICT** PICT format
- **PDF** PDF format
- **PNG** PNG format
- **Midi** MIDI format
- **EPSF** EPS format
- **TIFF** TIFF format
- **XML** Uncompressed MusicXML
- **MXL** Compressed MusicXML

So, to save a file using the current Sibelius file format, you would write `score.SaveAs("filename.sib", "SIBL");`

**SaveAsAudio** *(filename[, include all staves[, play from start]])*
Creates a WAV file (PC) or AIFF file (Mac) of the score, using Sibelius’s File > Export > Audio feature. If include all staves is True (the default), Sibelius will first clear any existing selection from the score so every instrument will be recorded; only selected staves will otherwise be exported. When play from start is True (also the default), Sibelius will record the entire score from beginning to end, otherwise from the current position of the playback line. Note that SaveAsAudio will only have an effect if the user’s current playback configuration consists of solely VST and/or AU devices. The functions returns True if successful, otherwise False (including if the user clicks Cancel during export).

SaveAsSibelius2(filename[, foldername])
Saves the score in Sibelius 2 format, overwriting any previous file with the same name. The optional foldername specifies the folder in which the file is to be saved. Note that saving as Sibelius 2 may alter some aspects of the score; see Sibelius Reference for full details.

SaveAsSibelius3(filename[, foldername])
Saves the score in Sibelius 3 format. See documentation for SaveAsSibelius2 above.

SaveAsSibelius4(filename[, foldername])
Saves the score in Sibelius 4 format. See documentation for SaveAsSibelius2 above.

SaveAsSibelius5(filename[, foldername])
Saves the score in Sibelius 5 format. See documentation for SaveAsSibelius2 above.

SaveAsSibelius6(filename[, foldername])
Saves the score in Sibelius 6 format. See documentation for SaveAsSibelius2 above.

SaveCopyAs(filename[, foldername])
Saves a copy of the score in the current version’s format without updating the existing score’s file name in Sibelius.

SetPlaybackPos(bar number, sr)
Sets the position of the playback line to a given bar number and rhythmic (sr) position.

StaveTypeId(stave type name)
Returns the identifier of the stave type with the given name, or the empty string if there is no such stave type.

SystemCount(page num)
The number of systems on a page (the first page of the score is page 1).

SymbolExists(symbol)
Returns True if the symbol index or name symbol is found in the score, otherwise False.

SymbolIndex(symbol name)
Returns the index of the symbol with the given name, or -1 if there is no such symbol.

TextStyleId(text style name)
Returns the identifier of the text style with the given name, or the empty string if there is no such text style.

ViewLiveTempo(view)
Switches View > Live Tempo on or off; set view to True to switch it on, or False to switch it off.

Variables

<table>
<thead>
<tr>
<th>Arranger</th>
<th>Arranger of score from File &gt; Score Info (read/write).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Artist</td>
<td>Artist of score from File &gt; Score Info (read/write)</td>
</tr>
<tr>
<td>Barlines</td>
<td>Returns a Barlines object containing information about the barline groupings in the score (read only).</td>
</tr>
<tr>
<td>BracketsAndBraces</td>
<td>Returns a BracketsAndBraces object containing information about the brackets and braces in the score (read only).</td>
</tr>
</tbody>
</table>
Object Reference

Composer
Composer of score from File > Score Info (read/write).

ComposerDates
Value of Composer's dates from File > Score Info (read/write).

Copyist
Copyist of score from File > Score Info (read/write).

Copyright
Copyright of score from File > Score Info (read/write).

CurrentDynamicPart
Returns or sets the current DynamicPart object for the Score (read/write). Sibelius will not automatically display the new part: use Sibelius.ShowDynamicPart() to change the displayed part.

CurrentPlaybackPosBar
Returns the bar number in which the playback line is currently located.

CurrentPlaybackPosSr
Returns the rhythmic position within the bar at which the playback line is currently located.

Dedication
Dedication of score from File > Score Info (read/write).

DocumentSetup
Returns a DocumentSetup object representing the settings in Layout > Document Setup (read only).

DynamicParts
Returns a DynamicPartCollection object representing the dynamic parts present in the Score. This object will always stay up to date, even if parts are added or deleted (read only).

EnableScorchPrinting
Corresponds to the Allow printing and saving checkbox in the Export Scorch Web Page dialog (read/write).

EngravingRules
Returns an EngravingRules object corresponding to selected settings in the House Style > Engraving Rules dialog (read only).

FileName
The filename for the score (read only).

FocusOnStaves
is True if View > Focus on Staves is switched on (read/write). See also Staff.ShowInFocusOnStaves.

HitPoints
The HitPointList object for the score (read/write).

InstrumentChanges
Value of Instrument changes from File > Score Info (read/write).

InstrumentTypes
Returns an InstrumentTypeList containing the score's instrument types, on which one may execute a for each loop to get information about each instrument type within the score.

IsDynamicPart
Returns True if the current active score view is a part (read only).

LiveMode
Is True (1) if Play > Live Playback is on (read/write).

Lyricist
Lyricist of score from File > Score Info (read/write).

MagneticLayoutEnabled
Returns True if the current score has Layout > Magnetic Layout switched on (read/write).

MainMusicFontName
Returns the name of the font specified as the Main music font (e.g. “Opus”, “Reprise”) in House Style > Edit All Fonts (read/write).

MainTextFontName
Returns the name of the font specified as the Main text font (e.g. “Times New Roman”, “Arial”) in House Style > Edit All Fonts (read/write).

MusicTextFontName
Returns the name of the font specified as the Music text font (e.g. “Opus Text”, “Reprise Text”) in House Style > Edit All Fonts (read/write).

NumberOfPrintCopies
The number of copies to be printed (read/write).

OpusNumber
Opus number of score from File > Score Info (read/write).

OriginalProgramVersion
The version of Sibelius in which this score was originally created, as an integer in the following format:

(major version) * 1000 + (minor version) * 100 + (revision) * 10

So Sibelius 3.1.3 would be returned as 3130.
Score

OtherInformation
More information concerning the score from File > Score Info (read/write).

PageCount
The number of pages in the score (read only).

PartName
Value of Part Name from File > Score Info (read/write).

Publisher
Publisher of score from File > Score Info (read/write).

Redraw
Set this to True (1) to make the score redraw after each change to it, False (0) to disallow redrawing (write only).

ScoreDuration
The duration of the score in milliseconds (read only).

ScoreEndTime
The duration of the score, plus the score start time (see above), in milliseconds (read only).

ScoreHeight
Height of a page in the score, in millimetres (read only).

ScoreStartTime
The value of Timecode of first bar, from Play > Video and Time > Timecode and Duration, in milliseconds (read only).

ScoreWidth
Width of a page in the score, in millimetres (read only).

Selection
The Selection object for the score, i.e. a list of selected objects (read only).

ShowMultiRests
Is True (1) if Layout > Show Multirests is on (read/write).

StaffCount
The number of staves in the score (read only).

StaffHeight
Staff height, in millimetres (read only).

Subtitle
Subtitle of score from File > Score Info (read/write).

SystemCount
The number of systems in the score (read only).

SystemObjectPositions
Returns a SystemObjectPositions object corresponding to the settings in House Style > System Object Positions for the score (read only).

SystemStaff
The SystemStaff object for the score (read only).

Title
Title of score from File > Score Info (read/write).

TransposingScore
Is True (1) if Notes > Transposing Score is on (read/write).

YearOfComposition
Value of Year of composition from File > Score Info (read/write).
Selection

for each variable in returns every BarObject (i.e. an object within a bar) in the selection.

for each type variable in produces each object of type type in the selection. Note that if the selection is a system selection (i.e. surrounded by a double purple box in Sibelius) then objects in the system staff will be returned in such a loop.

Methods

Clear()

Removes any existing selection(s) from the current active score.

ClipboardContainsData([clipboard Id])

Returns True if the given clipboard contains data. As with the Copy and Paste methods, 0 (or no arguments) refers to Sibelius's internal clipboard, and all other numeric values will interrogate the temporary clipboard with the matching ID.

Copy([clipboard Id])

Copies the music within the current selection to Sibelius's internal clipboard or a ManuScript-specific temporary clipboard, which goes out of scope along with the Selection object itself. If no clipboard Id is specified, or if it is set to 0, the selection will be copied to Sibelius's internal clipboard. Any other numeric value you pass in will store the data in a temporary clipboard adopting the ID you specify. Used in conjunction with Paste or PasteToPosition (see below).

Delete([remove staves])

Deletes the music currently selected in the active score. Akin to making a selection manually in Sibelius and hitting Delete. If remove staves is omitted or set to True, Sibelius will completely remove any wholly selected staves from the score. If you wish Sibelius to simply hide such staves instead, set this flag to False.

ExcludeStaff(staff number)

If a passage selection already exists in the current active score, an individual stave may be removed from the selection using this method.

IncludeStaff(staff number)

If a passage selection already exists in the current active score, a non-consecutive stave may be added to the selection using this method.

Paste([clipboard Id, reset positions])

Pastes the music from a given clipboard to the start of the selection in the current active score. If no clipboard Id is specified, or if it is set to 0, the data will be pasted from Sibelius's internal clipboard. Any other numeric value you pass in will take the data from a temporary clipboard you must have previously created with a call to Copy (see above). Returns True if successful. If reset positions is False, the positions of any objects that have been moved by the user in the source selection will be retained in the copy. This is the default behaviour. If you wish Sibelius to reset objects to their default positions, set this flag to True. This can be useful when copying one or more single objects (i.e. a non-passage selection).

Note that pasting into a score using this method will overwrite any existing music. Only one copy of the music will ever be made, so if your selection happens to span more bars or staves than is necessary, the data will not be duplicated to fill the entire selection area.

PasteToPosition(stave num, bar num, position[, clipboard Id[, reset positions]])

Pastes the music from a given clipboard to a specific location in the current active score. The optional parameters and pasting behavior works in the same way as calls to Paste.

RestoreSelection()

Restores the selection previously recorded with a call to StoreCurrentSelection. Usefully called at the end of a plug-in to restore the initial selection.

SelectPassage(start barNum[, end barNum[, top staveNum[, bottom staveNum[, start pos[, end pos]]]]])
Selection

Programmatically makes a passage selection around a given area of the current active score. When no end barNum is given, only the start barNum will be selected. If neither a top- nor bottom staveNum has been specified, every stave in the score will be selected, whereas if only a top staveNum has been supplied, only that one staff will be selected. Sibelius will begin the selection from the start of the first bar if no start pos has been given, similarly completing the selection at the end of the final bar if no end pos has been supplied.

NB: The start pos and end pos you supply may be altered by ManuScript: Sibelius requires a passage selection to begin and end at a NoteRest if it doesn't encompass the entire bar.

SelectSystemPassage(start barNum[, end barNum[, start pos[, end pos]]])

Programmatically makes a system selection around a given area of the current active score. When no end barNum is given, only the start barNum will be selected. Sibelius will begin the selection from the start of the first bar if no start pos has been given, similarly completing the selection at the end of the final bar if no end pos has been supplied.

NB: The start pos and end pos you supply may be altered by ManuScript: Sibelius requires a passage selection to begin and end at a NoteRest if it doesn't encompass the entire bar.

StoreCurrentSelection()

Stores the current selection in the active score internally. Can be retrieved with a call to RestoreSelection (see below). Useful called at the start of a plug-in to store the initial selection.

Transpose(degree, interval type[, keep double accs[, transpose keys]])

Transposes the currently selected music up or down by a specified degree and interval type. To transpose up, use positive values for degree; to transpose down, use negative values. Note that degrees are 0-based, so 0 is equal to a unison, 1 to a second and so on. For descriptions of the various available interval types, see Global constants on page 135. By default, Sibelius will transpose using double sharps and flats where necessary, but this behavior may be suppressed by setting the keep double accs flag to False. Sibelius will also transpose any key signatures within the selection by default, but can be overriden by setting the fourth parameter to False.

For help in calculating the interval and degree required for a particular transposition, see the documentation for the Sibelius.CalculateInterval and Sibelius.CalculateDegree methods.

Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BottomStaff</td>
<td>The number of the bottom staff of a passage (read only).</td>
</tr>
<tr>
<td>FirstBarNumber</td>
<td>The internal bar number of the first bar of a passage (read only).</td>
</tr>
<tr>
<td>FirstBarNumberString</td>
<td>The external bar number (including any bar number format changes) of the first bar of a passage (read only).</td>
</tr>
<tr>
<td>FirstBarSr</td>
<td>The position of the start of the passage selection in the first bar (read only).</td>
</tr>
<tr>
<td>IsPassage</td>
<td>True if the selection represents a passage, as opposed to a multiple selection (read only).</td>
</tr>
<tr>
<td>IsSystemPassage</td>
<td>True if the selection includes the system staff (read only).</td>
</tr>
<tr>
<td>LastBarNumber</td>
<td>The internal bar number of the last bar of a passage (read only).</td>
</tr>
<tr>
<td>LastBarNumberString</td>
<td>The external bar number (including any bar number format changes) of the last bar of a passage (read only).</td>
</tr>
<tr>
<td>LastBarSr</td>
<td>The position of the end of the passage selection in the last bar (read only).</td>
</tr>
<tr>
<td>TopStaff</td>
<td>The number of the top staff of a passage (read only).</td>
</tr>
</tbody>
</table>

Copying entire bars

Copying passages from one location in a score to another – or even from one score to another – is very simple. Here is an example function demonstrating how one might go about achieving this:

CopyBar(scoreSrc, barFirstSrc, barLastSrc, scoreDest, barFirstDest, barLastDest) // This is the function signature
Object Reference

{
    sel = scoreSrc.Selection;
    sel.SelectPassage(barFirstSrc.BarNumber, barLastSrc.BarNumber,
                     barFirstSrc.ParentStaff.StaffNum,
                     barLastSrc.ParentStaff.StaffNum);
    sel.Copy(0);
    selDest = scoreDest.Selection;
    selDest.SelectPassage(barFirstDest.BarNumber, barLastDest.BarNumber,
                          barFirstDest.ParentStaff.StaffNum,
                          barLastDest.ParentStaff.StaffNum);
    selDest.Paste(0);
}

Note that you may use any temporary clipboard or Sibelius's own internal clipboard if the source and destination locations are in the same score, however you can only use Sibelius's internal clipboard if the data is being transferred between two individual scores. This is because the temporary clipboards belong to the Selection object itself.

Copying multiple selections from one bar to another

Using a combination of the BarObject's Select method and the Selection object's Copy and PasteToPosition methods, it is possible to copy an individual or multiple selection from one location in a score to another. Bear in mind that Paste will always paste the material to the very start of the selection, so if you're copying a selection that doesn't start at the very beginning of a bar, you'll have to store the position of the first item and pass it to PasteToPosition when you later come to paste the music to another bar.

This example code below copies all items from position 256 or later from one bar to another. It is assumed that sourceBar is a valid Bar object, and destStaffNum and destBarNum contain the destination staff number and bar number respectively:

    sel = Sibelius.ActiveScore.Selection; // Get a Selection object for this score
    sel.Clear(); // Clear the current selection
    clipboardToUse = 1; // This clipboard ID we're going to use
    copyFromPos = 256; // Copy all objects from this point in the source bar
    posToCopyTo = 0; // Variable used to store the position of the first object copied
    for each obj in sourceBar { // Iterate over all objects in the bar
        if (obj.Position >= copyFromPos) { // Ignore objects before the start threshold
            obj.Select(); // Select each relevant object in turn
            if (posToCopyTo = 0) {
                posToCopyTo = obj.Position; // Remember the position of the first item
            }
        }
    }
    sel.Copy(clipboardToUse); // Copy the objects we've selected to the clipboard
    sel.PasteToPosition(destStaffNum, destBarNum, posToCopyTo, clipboardToUse); // And paste them to the destination bar at the relevant offset
Sibelius

There is a predefined variable that represents the Sibelius program. You can use the Sibelius object to open scores, close scores, display dialogs or (most commonly) to get currently open Score objects.

```python
for each variable in returns each open score.
```

**Methods**

`AppendLineToFile (filename, text[, use_unicode])`

Appends a line of text to the file specified (adds line feed). See comment for `AppendTextFile` above for explanation of the `use_unicode` parameter. Returns `True` if successful.

`AppendLineToRTFFile (filename, text)`

Appends a line of text to the file specified. Times New Roman 12pt is used, unless you specify a change of formatting. To change formatting, use the following backslash expressions:

- \B\ bold on
- \I\ italic on
- \U\ underline on
- \n\ new line
- \b\ bold off
- \i\ italic off
- \u\ underline off
- \f\fntname\ change to given font name (e.g. \fArial\ to switch to Arial)
- \s\ points\ set the font size to a specific point size (e.g. \s16\ to set the font to 16pts).

Note the difference in meaning of \s in the context of adding data to an RTF file, versus its use in the context of styling text directly within Sibelius (see Syntax on page 36 following).

`AppendTextFile (filename, text[, use_unicode])`

Appends text to the file specified. If the optional Boolean parameter `use_unicode` is `True`, then the string specified will be exported in Unicode format; if this parameter is `False` then it will be converted to 8-bit Latin-1 before being added to the text file. This parameter is `True` by default. Returns `True` if successful.

`CalculateDegree (source pitch, dest pitch, upward interval)`

Takes two note names in the form of a string (e.g. C, G#, Bb, Fx or Ebb) and a boolean that should be `True` if the interval you're wishing to calculate is upward. Returns a 0-based number describing the degree between the two notes. For example, `CalculateDegree("C#", "G", False)` would return 3.

`CalculateInterval (source pitch, dest pitch, upward interval)`

Takes two note names in the form of a string (e.g. C, G#, Bb, Fx or Ebb) and a boolean that should be `True` if the interval you're wishing to calculate is upward. Returns a number representing an Interval Type (see Global constants on page 135). You can use the value returned in calls to `NoteRest.Transpose` and `Selection.Transpose`. For example, `CalculateInterval("Bb", "G#", True)` would return `IntervalAugmented`.

`Close (show dialogs)`

Closes the current score or part view; if the current view is the last tab in the current window, the window will therefore also be closed. If the optional Boolean parameter is `True` then warning dialogs may be shown about saving the active score, and if it is `False` then no warnings are shown (and the score will not be saved).
Object Reference

**CloseAllWindows**(show dialogs)

Closes all open document windows. If the optional Boolean parameter is `True` then warning dialogs may be shown about saving any unsaved scores, and if it is `False` then no warnings are shown (and the scores will not be saved).

**CloseWindow**(show dialogs)

Closes the current window (i.e. closes all of the open tabs in the current window). If the optional Boolean parameter is `True` then warning dialogs may be shown about saving the score, and if it is `False` then no warnings are shown (and the score will not be saved).

**CreateFolder**(foldername)

Creates the folder of specified `foldername`; returns the Folder object created if successful, or null if it fails.

**CreateProgressDialog**(caption, min value, max value)

Creates the progress dialog, which shows a slider during a long operation.

**CreateRTFFile**(filename)

Creates the Rich Text Format (RTF) file specified. Any existing file with the same name is destroyed. Returns `True` if successful.

**CreateTextFile**(filename)

Creates the plain text file specified. Any existing file with the same name is destroyed. Returns `True` if successful.

**DestroyProgressDialog**()

Destroys the progress dialog.

**EnableControlById**(plugin, dialog, controlID, enable)

Dynamically enables or disables a given control on a plug-in dialog: `plugin` is a plug-in object, e.g. `Self`; `dialog` is a dialog object, and therefore should not be passed in quotation marks; `controlID` is the string corresponding to the control to be enabled or disabled; and `enable` is a Boolean parameter, which enables the control when set to `True` and disables the control when set to `False`.

**EnableNthControl**(nth control, enable)

Dynamically enables or disables a given control on a plug-in dialog. Can be called either before a dialog has been displayed (in which case the operation will apply to the next dialog you show), or while a dialog is already visible (in which case the operation will affect the top-most currently visible dialog).

Note that, using this method, controls can only be identified according to their order upon creation; for this reason, you are strongly recommended to use **EnableControlById**() instead. To find out the creation order, open the appropriate dialog in the plug-in editor, right click on the dialog's client area and choose `Set Creation Order` from the contextual menu that appears. Note that `nth control` expects a 0-based number, unlike the display given by **Set Creation Order**. By default, all controls will be enabled; to disable any given control, set `enable` to `false`.

**FileExists**(filename)

Returns `True` if a file exists or `False` if it doesn't.

**FolderExists**(foldername)

Returns `True` if a folder exists or `False` if it doesn't.

**GetDocumentsFolder**()

Returns the user's **My Documents** (Windows) or Documents (Mac) folder.

**GetElapsedTimeCentiseconds**(timer number)

Returns the time since `ResetStopWatch` was called for the given stop watch, in 100ths of a second.

**GetElapsedTimeMilliseconds**(timer number)

Returns the time since `ResetStopWatch` was called for the given stop watch, in 1000ths of a second.
**GetElapsedSeconds**(timer number)

Returns the time since **ResetStopWatch** was called for the given stop watch in seconds.

**GetFile**(file path)

Returns a new File object representing a file path e.g. `file=Sibelius.GetFile("c:\onion\foo.txt");`

**GetFolder**(file path)

Returns a new Folder object representing a file path e.g. `folder=Sibelius.GetFolder("c:\");`

**GetNotesForChord**(chord name)

Returns a ManuScript array giving the MIDI pitches corresponding to the named chord symbol.

**GetNotesForGuitarChord**(chord name)

Returns a ManuScript array giving the MIDI pitches and string numbers corresponding to the named guitar chord, using the most suitable fingering according to the user’s preferences. Strings are numbered starting at 0 for the bottom string and increasing upwards. The array returned has twice as many entries as the number of notes in the chord, because the pitches and string numbers are interleaved thus:

array[0] = MIDI pitch for note 0
array[1] = string number for note 0
array[2] = MIDI pitch for note 1
array[3] = string number for note 1
...

**GetScoresFolder()**

Returns a new Folder object representing the default Scores folder (as defined on the Files page of File ➤ Preferences).

**GetSyllabifier()**

Returns a new Syllabifier object, providing access to Sibelius’s internal syllabification engine.

**GetUserApplicationDataFolder()**

Returns the user’s Application Data (Windows) or Application Support (Mac) folder.

**GoToEnd()**

Moves the playback line to the end of the score.

**GoToStart()**

Moves the playback line to the start of the score.

**IsDynamicPartOpen**(dynamic part)

Returns True if the specified part and its corresponding Score is valid and is visible in a Score window within Sibelius.

**LiveTempoTap()**

Equivalent to tapping a beat during Live Tempo recording.

**MakeSafeFileName**(filename)

Returns a “safe” version of filename. The function removes characters that are illegal on Windows or Unix, and truncates the name to 31 characters so it will be viewable on Mac OS 9.

**MessageBox**(string)

Shows a message box with the string and an OK button.

**MoveActiveViewToBar**(bar number[, position])

Brings a given internal bar number into view. Has the same effect as Go to Bar in Sibelius. An optional position within the bar may also be specified, but if omitted, the very start of the bar will be brought into view.

**MoveActiveViewToSelection**(start of selection)
Brings the object(s) currently selected into view. If \textit{start of selection} is \texttt{False}, the end of the selection will be brought into view. If the optional argument is \texttt{True} or omitted, the start of the selection will be visible. Has the same effect as \texttt{Shift + Home/End} in Sibelius.

\textbf{New}([\texttt{manuscript paper}])

Creates and shows a new score. If the optional parameter manuscript paper is not supplied, Sibelius will create a blank score; manuscript paper should be the filename of the manuscript paper you want to create, minus its .sib file extension, optionally including the name of the category (subfolder) in which it is located, e.g. both \texttt{"String orchestra"} and \texttt{"Orchestral/String orchestra"} will work. Returns the score object corresponding to the new score.

\textbf{NthScore} (score index from 0)

Returns the $n$th open score (zero-based), or null if the specified index is not valid.

\textbf{Open} ([\texttt{filename [, quiet]}])

Opens and displays the given file. Filename must include its extension, e.g. \texttt{Song.sib}. If the optional boolean parameter \texttt{quiet} is set to \texttt{True}, then no error messages or dialogs will be displayed, even if the file could not be opened for some reason. Returns \texttt{True} if the file is opened successfully, \texttt{False} otherwise.

\textbf{Play}()

Plays the current score, from the current position of the playback line.

\textbf{PlayFromSelection}()

Plays from the current selection.

\textbf{PlayFromStart}()

Plays from the start of the score.

\textbf{PrependScreenreaderText} (\texttt{string})

Prepends \texttt{string} to the default screen reader description.

\textbf{Print} ([\texttt{number of copies [, dynamic part [, showdialog]]}])

Prints the specified number of copies of the current score or dynamic part using default settings. If \texttt{number of copies} is missing or a negative number, then the default number of copies for the score or part is printed, and if set to 0 no printing occurs. The optional \texttt{dynamic part} parameter must be a valid object of the active Score (this does not affect or use \texttt{Score.CurrentDynamicPart} for the Score printed); if it is not supplied, the active Score is printed instead. Returns \texttt{True} for success, \texttt{False} for failure. The second optional parameter, \texttt{showdialog}, is a Boolean: if set to \texttt{True}, Sibelius will show the \textit{Print} dialog, and if not specified or set to \texttt{False}, Sibelius will not show the dialog.

\textbf{PrintAllDynamicParts} ([\texttt{score}])

Prints the default number of copies of all dynamic parts, but does not print the full score. Prints the currently-active Score if the optional \texttt{score} parameter is not passed in. Returns \texttt{True} for success, \texttt{False} for failure.

\textbf{RandomNumber}()

Returns a random number.

\textbf{RandomSeed} (\texttt{start number})

Restarts the random number sequence from the given number.

\textbf{RandomSeedTime}()

Restarts the random number sequence based on the current time.

\textbf{RefreshDialog}()

Refreshes the data being displayed by any controls on the currently active plug-in dialog. For example, if a text object gets its string from a global variable and the value stored in this global variable has changed whilst the dialog is visible, calling \textbf{RefreshDialog} will update the text object on the dialog accordingly. Returns \texttt{True} if successful.

\textbf{ResetStopWatch} (\texttt{timer number})

Resets the given stop watch. \texttt{timer number} must be an integer greater than 0.
**ReadTextFile (filename, unicode)**

Reads the given filename into an array of strings, one per line. If the `unicode` parameter is true, the file is treated as Unicode, otherwise it is treated as ANSI (i.e. 8-bit) text, which is the default. The resulting array can be used in two ways:

```plaintext
lines = Sibelius.ReadTextFile("file.txt");
for each l in lines {
    trace(l);
}
```

or:

```plaintext
lines = Sibelius.ReadTextFile("file.txt");
for i=0 to lines.NumChildren {
    trace(lines[i]);
}
```

**ScreenreaderText (string)**

Replaces Sibelius's default screen reader description with `string`.

**SelectFileToOpen (caption, file, initial_dir, default extension, default type, default type description)**

Shows a dialog prompting the user to select a file to open. All parameters are optional. The method returns a file object describing the selection. For example:

```plaintext
file=Sibelius.SelectFileToOpen("Save Score","*.sib","c:\","sib","SIBE","Sibelius File");
```

Note that the `initial_dir` parameter has no effect on Mac, because it is unsupported by Mac OS X.

**SelectFileToSave (caption, file, initial_dir, default extension, default type, default type description)**

Shows a dialog prompting the user to select a file to save to. All parameters are optional. The method returns a file object describing the selection. File types and extensions:

<table>
<thead>
<tr>
<th>Description</th>
<th>Type</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMF graphics</td>
<td>&quot;EMF&quot;</td>
<td>emf</td>
</tr>
<tr>
<td>Windows bitmap</td>
<td>&quot;BMP&quot;</td>
<td>bmp</td>
</tr>
<tr>
<td>Macintosh PICT bitmap</td>
<td>&quot;PICT&quot;</td>
<td>pict</td>
</tr>
<tr>
<td>Sibelius score</td>
<td>&quot;SIBE&quot;</td>
<td>sib</td>
</tr>
<tr>
<td>MIDI file</td>
<td>&quot;Midi&quot;</td>
<td>mid</td>
</tr>
<tr>
<td>House style file</td>
<td>&quot;SIBS&quot;</td>
<td>lib</td>
</tr>
<tr>
<td>PhotoScore file</td>
<td>&quot;SCMS&quot;</td>
<td>opt</td>
</tr>
<tr>
<td>Web page</td>
<td>&quot;TEXT&quot;</td>
<td>html</td>
</tr>
<tr>
<td>TIFF graphics</td>
<td>&quot;TIFF&quot;</td>
<td>tif</td>
</tr>
<tr>
<td>PNG graphics</td>
<td>&quot;PNG&quot;</td>
<td>png</td>
</tr>
</tbody>
</table>

Note that the `initial_dir` parameter has no effect on Mac, because it is unsupported by Mac OS X.

**SelectFolder (caption)**

Allows the user to select a folder and returns a Folder object. The optional string parameter `caption` sets the caption of the dialog that appears.

**SetCurrentScoreViewType (view type)**

Allows plug-ins to switch between Panorama and normal view; values are `ViewTypePage` (0) and `ViewTypePanorama` (1).
Object Reference

**ShowDialog**(script name, object)

Shows a dialog from a dialog description and sends messages and values to the given object. Returns the value **True** (1) or **False** (0) depending on which button you clicked to close the dialog (typically **OK** or **Cancel**).

**ShowDynamicPart**(dynamic part[, newWindow])

Shows the specified dynamic part. The second optional Boolean parameter *newWindow* allows you to specify whether the part should open in a new tab (specify **False**, the default) or a new window (specify **True**). Returns **True** if the specified part can be shown, **False** otherwise. Can be used to bring a Score to the front by way of Sibelius.ShowDynamicPart(Score.CurrentDynamicPart).

**StartLiveTempoRecording()**

Starts recording Live Tempo; equivalent to choosing **Play** > **Record Live Tempo**.

**StopLiveTempoRecording()**

Stops recording Live Tempo.

**Stop()**

Stops the current score from playing.

**UpdateProgressDialog**(progress pos, status message)

Returns 0 if the user clicked **Cancel**.

**YesNoMessageBox**(string)

Shows a message box with **Yes** and **No** buttons. Returns **True** if **Yes** is chosen, else **False**.

**Variables**

**ActiveScore**

is the active Score object (read/write). Setting Sibelius.ActiveScore makes active the current dynamic part (which may be the full score rather than a part) of the score. If that window is not currently shown, a new window may be created according to the user’s preferences. Returns null if it fails to make the specified score or part active.

**ApplicationLanguage**

returns the language of the version of Sibelius currently running, always in English – e.g. **English**, **German**, **French** etc. (read only)

**ApplicationLanguageIsoString**

returns the two-letter ISO 3166 identifier of the language in which Sibelius is currently running, e.g. it *en, de, fr*, etc. (read only).

**CurrentTime**

returns a string containing the current time in the format hh:mm:ss, based on your own computer’s locale (read only)

**CurrentDateShort**

returns a string containing the current date in the format dd/mm/yyyy, based on your own computer’s locale (read only)

**CurrentDateLong**

returns a string containing the current date in the format dd MM yyyy, based on your own computer’s locale (read only)

**CurrentDate**

returns the current date and time as a DateTime object in local time (read only).

**HouseStyles**

the list of house styles available, as a ComponentList

**LocalizedApplicationLanguage**

returns the language in which Sibelius is currently running, in the localized language, e.g. it returns **Deutsch** when running in German (read only).

**ManuscriptPapers**

the list of manuscript papers available, as a ComponentList
OSVersionString

the current operating system in which the plug-in is running, as one of the following strings:

- Windows 95
- Windows 98
- Windows ME
- Windows NT 3.x
- Windows NT 4
- Windows 2000
- Windows XP
- Windows Vista
- Windows 7
- Mac OS X
- Mac OS X Jaguar
- Mac OS X Panther
- Mac OS X Tiger
- Mac OS X Leopard
- Mac OS X Snow Leopard
- Mac OS X Lion

If the operating system is unrecognized, the variable returns **Unknown system version**.

PathSeparator

returns the current path separator character (i.e. “\” on Windows, “/” on Mac).

Plugins

the list of plug-ins installed. See the documentation for the Plugin object.

Playing

is True if a score is currently being played (read only).

ProgramVersion

the current version of Sibelius in which the plug-in is running, as an integer in the following format:

(major version) * 1000 + (minor version) * 100 + (revision) * 10

So Sibelius 3.1.3 would be returned as **3130**.

ScoreCount

is the number of scores being edited (read only).

ViewHighlights

is True if **View ➤ Highlights** is switched on (read/write).

ViewNoteVelocities

is True if **View ➤ Live Playback Velocities** is switched on (read/write).

ViewNoteColors

the current **View ➤ Note Colors** setting used (read/write).

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>0</td>
</tr>
<tr>
<td>Notes out of Range</td>
<td>1</td>
</tr>
<tr>
<td>Pitch Spectrum</td>
<td>2</td>
</tr>
<tr>
<td>Voice Colors</td>
<td>3</td>
</tr>
</tbody>
</table>
SparseArray

For more information about using sparse arrays in ManuScript, see Sparse arrays on page 21.

To create a sparse array, use the built-in method `CreateSparseArray(a1, a2, a3, a4...an)`.

`for each` allows you to iterate over the contents of a sparse array.

**Methods**

**Concat(array1, array2 ... arrayN)**

Concatenate zero or more sparse arrays to this one, and return it as a one-level deep copy (so if a sparse array contains other arrays, for example, then the new sparse array will contain references to those arrays, not copies of them). This method does not modify the original sparse array.

**Join([separator])**

Returns the array as a string, with each populated element separated by the optional `separator`. If you don't specify `separator`, the default separator is a comma.

**Push(value1, value2, value3 ... valueN)**

Pushes one or more values to the end of the array.

**Pop()**

Returns the last element of the array, and removes it from the array.

**Reverse()**

Reverses the sparse array in place, modifying the sparse array being operated on. The reversed array only populates the elements needed to create the reversed array.

**Slice(start[, end])**

Returns a new sparse array of the elements starting from `start` and up to, but not including, the optional `end`. `start` and `end` can be negative indices referring to offsets from the end of the array.

**Variables**

**Length**

Returns or sets the length of the array (read/write).

**ValidIndices**

Returns a sparse array containing only the populated indices of the original sparse array, i.e. those that are not null.

**Converting old-style arrays to new sparse arrays**

The SparseArray object is a replacement for the old Array object, which was a more limited kind of array that could only hold strings and integers, but no other kind of objects. You are recommended to use the new SparseArray object for all arrays in your plug-ins, but if you have an existing plug-in in which old-style Arrays are used, you can convert them to SparseArrays as follows:

`Array.ConvertToSparseArray()` returns a new SparseArray object, populated with strings converted from the old-style Array.
SpecialBarline

Derived from a BarObject
These can only be found in system staves.

Methods
None.

Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BarlineType</td>
<td>The name of the type of special barline, expressed as a string.</td>
</tr>
<tr>
<td>BarlineInternalType</td>
<td>The type of the barline, expressed as a numeric ID which maps to one of the SpecialBarline global constants (see Global constants on page 135).</td>
</tr>
</tbody>
</table>
Staff

These can be normal staves or the system staff. The system staff contains objects that apply to all staves, such as SpecialBarlines and text using a system text style.

A Staff contains Bar objects.

for each variable in returns each object in the staff.

for each type variable in returns each item of type type in the staff in chronological order (i.e. in order of rhythmic position in each bar).

Methods

AddClef (pos, concert pitch clef, transposed pitch clef)

Adds a clef to the staff at the specified position. concert pitch clef determines the clef style when Notes Transposing Score is switched off: the optional transposed pitch clef parameter determines the clef style when this is switched on. Clef styles should be an identifier like “clef.treble”; for a complete list of available clef styles, see Clef styles on page 139. Alternatively you can give the name of a clef style, e.g. “Treble,” but bear in mind that this may not work in non-English versions of Sibelius.

AddLine (pos, duration, line style, [dx, dy, [voicenumber, [hidden]]])

Adds a line to staff (please see the documentation in Bar object below).

AddNote (pos, sounding pitch, duration, [tied [, voice [, diatonic pitch[, string number]]]])

Adds a note to staff, adding to an existing NoteRest if already at this position (in which case the duration is ignored); otherwise creates a new NoteRest. Will add a new bar if necessary at the end of the staff. The position is in 1/256th quarters from the start of the score. The optional tied parameter should be True if you want the note to be tied. Voice 1 is assumed unless the optional voice parameter (with a value of 1, 2, 3 or 4) is specified. You can also set the diatonic pitch, i.e. the number of the “note name” to which this note corresponds, 7 per octave (35 = middle C, 36 = D, 37 = E and so on). If a diatonic pitch of zero is given then a suitable diatonic pitch will be calculated from the MIDI pitch. The optional string number parameter gives a string number for this note, which is only meaningful if the note is on a tablature stave. If this parameter is not supplied then a default string number is calculated based on the current tablature stave type and the guitar tab fingering options (specified on the Note Input page of File ➤ Preferences). Returns the Note object created (to get the NoteRest containing the note, use Note.ParentNoteRest).

When adding very short notes to tuplets, Sibelius may be unable to find a legal place for the note in the bar. Should this happen, Sibelius will return null. You should therefore check for a valid object if there is any likelyhood that this situation may arise in your code.

N.B.: If you add a note to a score that intersects an existing tuplet, Sibelius will try to snap the note to the closest sensible place within that tuplet. However, you are advised to use Tuplet.AddNote() for this purpose as it is void of any ambiguity.

AddStaffAbove (ossia, [start bar number[, end bar number[, start pos[, end pos]]]])

Adds a new staff above the staff. Set ossia to True to create an ossia (small) staff. The other, optional parameters determine where the staff should be visible: if you do not specify a start bar number, the staff will be visible from the start of the score; if you do not specify an end bar number, the staff will be visible to the end of the score. If you specify a start and/or end bar number, the staff will be hidden outside that range by way of an instrument change to the No instrument (hidden) instrument type. start pos and end pos represent the rhythmic position within the start bar number and end bar number respectively, and if not specified, start pos will default to the start of the bar, and end pos will default to the end of the bar. Returns the staff created, or null if the call fails.

AddStaffBelow (ossia, [start bar number[, end bar number[, start pos[, end pos]]]])

Adds a new staff below the staff. See AddStaffAbove() above for details.

AddSymbol (pos, symbol index or name)

Adds a symbol to staff (please see the documentation in Bar object below).
**CurrentKeySignature** *(bar number)*

Returns a KeySignature valid at the bar number passed.

**NthBar** *(n)*

Returns the *n*th bar in the staff, counting from 1.

**ResetSpaceAroundStaff** *(above, below[, from bar[, to bar]])*

Does the equivalent of Layout  Reset Space Above Staff and/or Reset Space Below Staff for the given range of bars in a staff. Set *above* to True to reset the space above the staff, and *below* to True to reset the space below the staff. If *from bar* is not specified, Sibelius sets it to 1; if *to bar* is not specified, Sibelius sets it to the last bar of the score.

**SetSound** *(styleID[, set SoundStage])*

Changes the initial playback sound of this staff to be the default sound for the given default instrument `styleID`. For a complete list of default instrument style IDs in Sibelius, see Instrument types on page 139. If the optional Boolean parameter is set to False, then the SoundStage information (volume, pan and distance) for this staff will be unchanged. If it is omitted or set to True, then the SoundStage information will be set to the default for the new sound.

**SetSoundID** *(soundID)*

Changes the initial playback sound of this staff to the given `soundID`.

**Staff** *[array element]*

Returns the *n*th bar (counting from 1) e.g. `Staff[1]`.

### Variables

**BankHigh**

Controls MIDI controller 0, used to select the “coarse” bank number for this stave, and corresponding to the Mixer control of the same name. The range is 0–127, or –1 if you don’t want to send this controller message at the start of playback. Note that not all MIDI devices support multiple banks (read/write).

**BankLow**

Controls MIDI controller 32, used to select the “fine” bank number for this stave, and corresponding to the Mixer control of the same name. The range is 0–127, or –1 if you don’t want to send this controller message at the start of playback. Note that not all MIDI devices support multiple banks (read/write).

**BarCount**

Number of bars in the staff (read only).

**Channel**

The MIDI channel number of this staff, numbered 1–16 (read/write).

**Distance**

The reverb “distance” of this staff, corresponding to the control of the same name in the Mixer. This is a percentage, used to scale the overall reverb settings from the Performance dialog (read/write).

**FullInstrumentName**

Gives the full instrument name of the staff, empty for an unnamed staff (read/write).

**FullInstrumentNameWithFormatting**

Gives the full instrument name of the staff including any changes of font or style, if any (read/write).

**NumStavesInSameInstrument**

The number of staves belonging to the default instrument from which this staff was created (read only).

**InitialClefStyle**

The name of the initial clef on a staff, depending on the state of Notes  Transposing Score (read only).

**InitialClefStyleId**

The style identifier of the initial clef on a staff, depending on the state of Notes  Transposing Score (read only).

**InitialInstrumentType**

Returns an InstrumentType object for the instrument type at the start of the staff.

**InitialKeySignature**

Returns the KeySignature object at the start of this staff (read only).
<table>
<thead>
<tr>
<th>Object Reference</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InitialStyleId</td>
<td>Returns the style identifier of the staff (read only). To create an instrument from such an ID, pass the style as the first argument to Score.CreateInstrument. For a complete list of all the default instrument names in Sibelius, see Instrument types on page 139.</td>
</tr>
<tr>
<td>InstrumentName</td>
<td>Gives the full instrument name of the staff in the form that is displayed on the Instruments and Staves dialog in Sibelius (read only). For an unnamed stave, this will be “[Piano]” for example, where Piano is the default instrument name of the stave (see below). To get the internal name (which will be empty for unnamed staves), use the read/write variables FullInstrumentName or ShortInstrumentName instead.</td>
</tr>
<tr>
<td>IsSystemStaff</td>
<td>True or False depending on whether this staff is a system staff or not (read only).</td>
</tr>
<tr>
<td>IsVocalStaff</td>
<td>Returns True if the instrument type used by the staff has the Vocal staff option switched on, meaning that e.g. the default positions of dynamics should be above the staff rather than below (read only).</td>
</tr>
<tr>
<td>MuteMode</td>
<td>Specifies whether or not this stave will play back. Corresponds to the mute button in the Mixer. The supported values are defined as global constants (see Global constants on page 135) and are Muted, HalfMuted and NotMuted (read/write).</td>
</tr>
<tr>
<td>Pan</td>
<td>The MIDI stereo panning position of this staff (corresponding to the pan control in the Mixer). Permissible values are –100 to 100, with positive values being to the right and negative to the left (read/write).</td>
</tr>
<tr>
<td>ParentScore</td>
<td>Returns the staff’s parent Score object (read only).</td>
</tr>
<tr>
<td>ShortInstrumentName</td>
<td>Gives the short instrument name of the staff, empty for an unnamed staff (read/write).</td>
</tr>
<tr>
<td>ShortInstrumentNameWithFormatting</td>
<td>Gives the short instrument name of the staff including any changes of font or style, if any (read/write).</td>
</tr>
<tr>
<td>ShowInFocusOnStaves</td>
<td>If True then this staff will be shown when Layout &gt; Focus on Staves is switched on (see also Score.FocusOnStaves). This variable cannot be set to False unless it is also True for at least one other staff in the score (read/write).</td>
</tr>
<tr>
<td>Solo</td>
<td>True or False depending on whether this staff plays back in “solo” mode, corresponding to the Mixer button of the same name (read/write).</td>
</tr>
<tr>
<td>SoundIdOverrideIfAny</td>
<td>Returns a string containing the sound ID override set in the mixer for the staff. If no override has been set, an empty string is returned (read only).</td>
</tr>
<tr>
<td>Small</td>
<td>True if the staff is small (e.g. an ossia staff), False if it is normal sized (read/write).</td>
</tr>
<tr>
<td>StaffNum</td>
<td>Returns the number of this stave, counting from 1 at the top of the currently-viewed part. Returns 0 for for SystemStaff objects (read only).</td>
</tr>
<tr>
<td>Volume</td>
<td>The overall MIDI volume of this staff, corresponding to its fader in the Mixer. Permissible values are 0–127 (read/write).</td>
</tr>
</tbody>
</table>
Syllabifier

Acts as a wrapper around Sibelius's internal Syllabification engine, exposing its functionality to ManuScript.

Methods

AbbreviateUsingApostrophe

When the abbreviate flag is set to True when calling Syllabify, Sibelius will replace vowels that have been combined with the previous syllable with an apostrophe if this option is switched on – e.g. Vege-ta-bles vs Veg'-ta-bles. Calling this method will cause the syllabification engine to recalculate its result if necessary.

GetNthSyllable

Once a string has been syllabified by calling the Syllabify method, you can use this method to return each individual syllable as a string

NthSyllableEndsWord

Once a string has been syllabified by calling the Syllabify method, you can use this method to find out whether each syllable occurs at the end of a word

Syllabify

Breaks a string down into its syllabic components, returning the number of syllables in the resultant syllabification, or 0 if an error has occurred. The rules of the specified language will be used, and you may legally supply either a language ID, or the localized language name. To get the individual syllables, you should call the GetNthSyllable and NthSyllableEndsWord methods documented below.

If the language argument is omitted, Sibelius will attempt to automatically identify the language of the text. If this is not possible, or if an unrecognised language ID or name has been supplied, 0 will be returned.

When abbreviate is True, each ambiguous word in the string will be syllabified using the minimal number of syllables. For example, syllabifying "Everybody likes vegetables" would return "Eve-ry-bod-y likes vege-ta-bles" with this flag set to True, otherwise "E-ve-ry-bod-y likes veg-e-ta-bles".

Variables

AbbreviateUsingApostrophe

Returns True/False depending on whether the syllabification engine is set to abbreviate combined syllables with an apostrophe (read only – call method with same name for write access)

AvailableLanguageIds

Returns an array containing a list of the available syllabification languages as three-letter non-translatable IDs – e.g. ENG (English), GER (German), LAT (Latin). These IDs are identical in all localized versions of Sibelius (read only)

AvailableLanguages

Returns an array containing a list of the available syllabification languages as localized strings (read only)

NumberOfSyllables

Returns the number of syllables in the hyphenated string generated by calling the Syllabify method (read only)

SyllabifiedString

Returns the resultant hyphenated string generated by calling the Syllabify method (read only)
SymbolItem and SystemSymbolItem

Derived from a BarObject. For system symbols (i.e. symbols belonging to the system staff, retrieved with for each on the system staff object), the type of symbol objects is SystemSymbolItem, not SymbolItem.

Methods
None.

Variables

- **Index**  
  The index of this symbol in the list of symbols. This corresponds to its position in the Create Symbol dialog, counting from zero left-to-right and top-to-bottom (read only).

- **Name**  
  The name of this symbol. May be translated in non-English language versions of Sibelius (read only).

- **Size**  
  The draw size of the symbol, corresponding to the four available options in the Symbols dialog in Sibelius. The four available values are NormalSize, CueSize, GraceNoteSize and CueGraceNoteSize, all defined as global constants (read/write).
SystemObjectPositions

Accessed from a Score object. Corresponds to the settings in House Style > System Object Positions.

Methods

GetNthStaffShowsSystemObjects(staffNum)

Returns True if the given staff number staffNum (relative to the current part) is showing system objects above it, otherwise False.

SetNthStaffShowsSystemObjects(staffNum, show)

Tells the staff with the given staff number staffNum (relative to the current part) either to show or not show system objects above it. This will have no effect if you pass in the top staff in the part, or if the maximum number of staves allowed to show system objects has already been met.

Clear([removeBelowBottomStaff])

Allows you to clear all the system object positions (apart from the compulsory one above the top staff) in a single operation; set the optional Boolean parameter removeBelowBottomStaff to True to also clear the Below bottom staff system object position.

Variables

NumStavesShowingSystemObjects

Returns the current number of staves showing system object positions (read only)

ShowSystemObjectsBelowBottomStaff

Returns True if system objects should show below the bottom staff, otherwise False (read/write).
### Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IsALine</strong></td>
<td>Returns true if the object is a line object. (Note that this is a variable, not a method, unlike the <code>IsObject()</code> method for all objects.)</td>
</tr>
<tr>
<td><strong>Type</strong></td>
<td>A string giving the name of the type of an object. The strings for the first 4 types above are &quot;SystemStave&quot;, &quot;Stave&quot;, &quot;MusicSelectionList&quot;, and &quot;Bar&quot;. Note that this variable is also a member of all objects that occur in bars.</td>
</tr>
</tbody>
</table>
SystemStaff

There is one SystemStaff object per score. The SystemStaff contains objects which apply to all staves, such as SpecialBarlines and text using a system text style. Unlike normal staves, the SystemStaff does not appear in the score itself. As such, most of the variables and methods supported for Staff objects are not available on a SystemStaff. Those that are supported by SystemStaff are as follows.

Methods

**CurrentKeySignature**(bar number)
- Returns a KeySignature valid at the bar number passed.

**CurrentTimeSignature**(bar number)
- Returns a TimeSignature valid at the bar number passed.

**NthBar**(n)
- Returns the n-th bar in the staff, counting from 1.

**SystemStaff**(array element)
- Returns the n-th bar (counting from 1) e.g. `SystemStaff[1].`

Variables

**BarCount**
- Number of bars in the staff (read only).

**InitialKeySignature**
- Returns the KeySignature object at the start of this staff (read only).

**IsSystemStaff**
- Returns **True** for a SystemStaff (read only).
Object Reference

Text and SystemTextItem

Derived from a BarObject. For system text (i.e. text belonging to the system staff, retrieved with for each on the system staff object), the type of text objects is SystemTextItem, not Text.

Methods
None.

Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>StyleAsText</td>
<td>The text style name (read/write).</td>
</tr>
<tr>
<td>StyleId</td>
<td>The identifier of the text style of this piece of text (read/write).</td>
</tr>
<tr>
<td>Text</td>
<td>The text as a string (read/write).</td>
</tr>
<tr>
<td>TextWithFormatting</td>
<td>Returns an array containing the various changes of font or style (if any) within the string in a new element (read only). For example, &quot;This text is \B\bold\b, and this is \italic\i&quot; would return an array with eight elements containing the following data:</td>
</tr>
</tbody>
</table>

```plaintext
arr[0] = "This text is "
arr[1] = "\B\"
arr[2] = "bold"
arr[3] = "\b\"
arr[4] = ", and this is "
arr[5] = "\I\"
arr[6] = "italic"
arr[7] = "\i\"
```

TextWithFormattingAsString | The text including any changes of font or style (read only). |
TimeSignature

Derived from a BarObject.

Methods
None

Variables
- **AllowCautionary**
  Returns `True` if the time signature is set to show a cautionary at the end of the previous system, if it occurs at the start of a system (read/write).
- **Denominator**
  The time signature's bottom number (read only).
- **Numerator**
  The time signature's top number (read only).
- **Text**
  The time signature as text. You can use this to detect common time and alla breve time signatures by comparing it to the global constants `CommonTimeString` and `AllaBreveTimeString`, which define the Unicode characters used by these symbols. Other time signatures will be of the form “4/4” (read only).
TreeNode

These are used internally by ManuScript to implement arrays and hashes (returned with the CreateArray and CreateHash methods), and to represent global data (defined in the plugin editor). Each TreeNode can contain a label, a piece of data and a list of “children,” which are also TreeNodes. Normally, any access to a TreeNode object will access the data that is held, so that you don’t need to know anything about them, but there are also some extra variables and methods that may be useful in some circumstances. These can be called on any array, hash or global variable, and on any member of such a structure.

Methods

WriteToString

Returns a string that represents the structure of this TreeNode object. In this representation, the data of a TreeNode is surrounded by double quotes and the label is not. Note that a label need not be defined. Any children of the TreeNode (also TreeNode objects themselves) are contained within curly braces { and }. To obtain child TreeNodes, use the normal array operator, as described in the documentation for arrays and hashes.

Variables

Label

The label of this TreeNode.

NumChildren

The number of child TreeNodes belonging to this TreeNode object.
Tuplet

Derived from a BarObject.

**Methods**

**AddNestedTuplet** *(posInTuplet, left, right, unit[, style[, bracket[, fullDuration]]])*

Nests a new tuplet bracket within an existing tuplet at a position relative to the duration and scale-factor of the existing tuplet. The *left* and *right* parameters specify the ratio of the new tuplet, e.g. 3 (left) in the time of 2 (right). The *unit* parameter specifies the note value (in 1/256th quarters) on which the tuplet should be based. For example, if you wish to create an eighth note (quaver) triplet group, you would use the value 128. The optional *style* and *bracket* parameters take one of the pre-defined constants that affect the visual appearance of the created tuplet; see **Global constants** on page 135. If *fullDuration* is true, the bracket of the tuplet will span the entire duration of the tuplet. Returns the Tuplet object created.

NB: If **AddNestedTuplet**() has been given illegal parameters, it will not be able to create a valid Tuplet object. Therefore, you should test for inequality of the returned Tuplet object with *null* before attempting to use it.

**AddNote** *(posInTuplet, pitch, duration[, tied[, diatonic pitch[, string number]]])*

Adds a note to an existing tuplet, adopting the same voice number as used by the tuplet itself. Please note that *posInTuplet* is relative to the duration and scale-factor of the tuplet bracket itself. Therefore, if you wanted to add a quarter note/crotchet to the second beat of a quarter note/crotchet triplet, you would simply use the value 256 – not 341!

**utils.SplitTuplet** *(tuplet, splitpoint)*

Split the tuplet object *tuplet* at the specified *splitpoint*, which is a number in relation to the tuplet's parent bar. It then splits a nest of tuplets at that point in the bar. This method is provided by the **utils.plg** – see **Utils** on page 126.

**Variables**

**Bracket**

The bracket type of the tuplet (e.g. none, auto; see **Global constants** on page 135).

**FullDuration**

True if the bracket of the tuplet spans its entire duration.

**Left**

The left side of the tuplet, e.g. 3 in 3:2 (read only).

**ParentTupletIfAny**

If the tuplet intersects a tuplet, the innermost Tuplet object at that point in the score is returned. Otherwise, *null* is returned (read only).

**PlayedDuration**

The true rhythmic duration of the tuplet e.g. for crotchet triplet this would be the duration of a minim (read only).

**PositionInTuplet**

Returns the position of the tuplet relative to the duration and scale-factor of its parent tuplet. If the tuplet does not intersect a tuplet, its position within the parent Bar is returned as usual (read only).

**Right**

The rightside of the tuplet, e.g. 2 in 3:2 (read only).

**Style**

The style of the tuplet (e.g. number, ratio, ratio + note; see **Global constants** on page 135).

**Text**

The text shown above the tuplet (read only).

**Unit**

The unit used for the tuplet, e.g. 256 for a triplet of quarter notes (read only).
Sibelius installs a plug-in called **utils.plg** that contains a set of useful and common methods that can be called directly by other plug-ins. It is not intended to be run as a plug-in in its own right, so does not appear in the **Plug-ins** menu.

The methods available via **utils.plg** are as follows:

- **utils.AbsoluteValue(value)**
  Returns the absolute value of a number, i.e. its numerical value without regard to its sign.

- **utils.AddFractions(x, y)**
  Adds two fractions \( x \) and \( y \), passed in as ManuScript arrays. Returns an array with the result of the addition.

- **utils.Bstring(x)**
  Returns a binary string (e.g. "101010") equivalent to the number \( x \).

- **utils.bwAND(x, y)**
  Equivalent to the C++ bitwise AND (\&) operator. For example, **utils.bwAND(129, 1)** is equal to 1.

- **utils.bwOR(x, y)**
  Equivalent to the C++ bitwise inclusive OR (\|) operator. For example, **utils.bwOR(64, 4)** is equal to 68.

- **utils.bwXOR(x, y)**
  Equivalent to the C++ bitwise exclusive XOR (^) operator. For example, **utils.bwXOR(4, 6)** is equal to 2.

- **utils.CapableOfDeletion()**
  Returns **True** if the object can be deleted using **Delete()**, which is determined by checking Sibelius's version number.

- **utils.CaseInsensitiveComparison(s1, s2)**
  Returns **True** if the two strings \( s1 \) and \( s2 \) match, ignoring case.

- **utils.CastToBool(x)**
  Returns the variable \( x \) explicitly cast as a Boolean.

- **utils.CastToInt(x)**
  Returns the variable \( x \) explicitly cast as an integer.

- **utils.CastToStr(x)**
  Returns the variable \( x \) explicitly cast as a string.

- **utils.CombineArraysOfBool(arr1, arr2)**
  Concatenates two arrays containing Boolean values and returns the result.

- **utils.CombineArraysOfInt(arr1, arr2)**
  Concatenates two arrays containing integral values and returns the result.

- **utils.CombineArraysOfString(arr1, arr2)**
  Concatenates two arrays containing string values and returns the result.

- **utils.CopyTextFile(source, dest)**
  Copies an existing text file from one location to another, returning **True** if successful.

- **utils.CreateArrayBlanket(value, size)**
  Returns an array with \( size \) elements, each containing a blanket value specified by the first parameter.
**Utils**

`utils.DeleteStaff(score, nth staff, retain selection)`

Deletes an entire staff and its content from a given score, returning `True` if successful. If `retain selection` is `True`, Sibelius will ensure any item(s) that were selected prior to the staff’s deletion are still selected.

`utils.DenaryValue(x)`

Returns a number in base 10 equivalent to binary number `x`, which must be provided as a string.

`utils.DivideFractions(x, y)`

Divides fraction `x` by fraction `y`, passed in as ManuScript arrays. Returns an array with the result of the division.

`utils.ExtractFileName(filename)`

Returns just the filename portion of a string `filename` containing both a path and a filename.

`utils.Format(str, [val1, val2, val3 ...])`  
Provides a simple means of replacing human-readable data types in a string. Each successive instance of `%s` in `str` is replaced with the value of the next remaining unused argument. e.g. `s = utils.Format("The %s brown %s jumps %s the lazy %s", "quick", "fox", "over", "dog");`

`utils.FormatTime(ms)`

Formats a time, given in milliseconds, to a human-readable string using the format `mm'ss.z` (where `z` is centiseconds).

`utils.FractionAsDecimal(x)`

Returns the decimal equivalent of the fraction `x`, which is passed in as an array.

`utils.FractionDenominator(x)`

Returns the denominator of fraction `x`, which is passed in as an array.

`utils.FractionNumerator(x)`

Returns the numerator of fraction `x`, which is passed in as an array.

`utils.GetAppDir()`

Returns the path of the Sibelius executable as a string.

`utils.GetArrayIndex(arr, value)`

Returns the index of `value` in the array `arr`, or `-1` if it doesn’t exist in the array.

`utils.GetBits(x)`

Returns an array containing the list of powers of two whose cumulative sum equates to the value of `x`.

`utils.GetGlobalApplicationDataDir()`

Returns the path of the system’s global application data area as a string.

`utils.GetLocationTime(score, barNum, position)`

Returns the precise time (in milliseconds) of a given location in a score. The position should be local to the start of the bar number you have supplied. Use the utils library to achieve this if your plug-in needs to be backwards compatible with Sibelius 4; otherwise call the Score object’s function with the same name.

`utils.GetMillisecondsFromTime(time)`

If you pass in a time expressed in milliseconds (e.g. one minute being 60,000), this function returns the milliseconds portion of the number (in this case 60,000 modulus 1000 = 0).

`utils.GetMinutesFromTime(time)`

If you pass in a time expressed in milliseconds, this function returns the minutes portion of the number (e.g. if `time = 120,262` milliseconds, this function returns 2).

`utils.GetObjectTime(score, obj)`
Returns the precise time (in milliseconds) that the object \( \text{obj} \) occurs from the start of a given score, taking into account tempo changes, performance markings and any other events in the score that have an effect on playback. Use this method to achieve this if your plug-in needs to be backwards compatible with Sibelius 4; otherwise use the \textbf{Time} property of the \texttt{BarObject} object whose time you wish to determine.

\begin{verbatim}
utils.GetPluginId(\text{plug-in})
\end{verbatim}

This enables you to identify a plug-in by entering the line of code \texttt{PluginUniqueID = "someUniqueId";} in a plug-in's \texttt{Initialize} method. When you pass a plug-in object to this function, it scans the plug-in's code and returns its unique ID if it has one, otherwise an empty string.

\begin{verbatim}
utils.GetSibeliusPluginsFolder()
\end{verbatim}

This is a wrapper around the deprecated \texttt{GetPluginsFolder()} function, and returns the path of the \texttt{Plugins} folder.

\begin{verbatim}
utils.GetSibMajorVersion()
\end{verbatim}

Returns the major version number of Sibelius.

\begin{verbatim}
utils.GreatestCommonDivisor(m,n)
\end{verbatim}

Returns the greatest common divisor of two non-zero integers, i.e. the largest positive integer that divides both numbers without remainder.

\begin{verbatim}
utils.IsInArray(arr, value)
\end{verbatim}

Returns \texttt{True} if \texttt{value} exists in the array \texttt{arr}.

\begin{verbatim}
utils.IsNumeric(str[, integer only])
\end{verbatim}

Returns \texttt{True} if the string \texttt{str} is numeric. Set the optional Boolean parameter \texttt{integer only} to \texttt{True} if you want the method to only return \texttt{True} if \texttt{str} is an integer (so that you can disallow floating point numbers).

\begin{verbatim}
utils.LowerCase(str)
\end{verbatim}

Returns the ANSI string \texttt{str} in lowercase.

\begin{verbatim}
utils.MakeFraction(x,y)
\end{verbatim}

Creates a fraction with \texttt{x} as the numerator and \texttt{y} as the denominator. The fraction is returned as a normal ManuScript array. (Manipulating fractions means you never have to worry about rounding errors.)

\begin{verbatim}
utils.max(x, y)
\end{verbatim}

Returns the greater of two numbers.

\begin{verbatim}
utils.min(x, y)
\end{verbatim}

Returns the lesser of two numbers.

\begin{verbatim}
utils.MultiplyFractions(x,y)
\end{verbatim}

Multiplies fraction \texttt{y} by fraction \texttt{x}, passed in as ManuScript arrays. Returns an array with the result of the multiplication.

\begin{verbatim}
utils.PatternCount(pattern, str)
\end{verbatim}

Returns the number of times the substring \texttt{pattern} exists in \texttt{str}.

\begin{verbatim}
utils.Pos(subStr, str)
\end{verbatim}

Returns the zero-based position of the first instance of the sub-string \texttt{subStr} in \texttt{str}, or \texttt{-1} if it isn't found.

\begin{verbatim}
utils.PosReverse(subStr, str)
\end{verbatim}

Returns the zero-based position of the last instance of the sub-string \texttt{subStr} in \texttt{str}, or \texttt{-1} if it isn't found.

\begin{verbatim}
utils.RaisePower(x, y)
\end{verbatim}

Raises \texttt{x} to the \texttt{y}th power, where \texttt{y} is a positive integer.

\begin{verbatim}
utils.Replace(inStr, toFind, replaceWith, replaceAll)
\end{verbatim}

Replaces a sub-string in a string with a new value. It looks for \texttt{toFind} in the string \texttt{inStr}, and if it finds it, replaces it with \texttt{replaceWith}. If the Boolean \texttt{replaceAll} is \texttt{False}, it only changes the first instance found; if it's \texttt{True}, it replaces all instances.
Utils

**utils.ReverseArrayOfBool (arr)**

Reverses the order of the elements in an array of Booleans.

**utils.ReverseArrayOfInt (arr)**

Reverses the order of the elements in an array of integers.

**utils.ReverseArrayOfString (arr)**

Reverses the order of the elements in an array of strings.

**utils.RoundToNDecimalPlaces (number, precision)**

Returns a string containing the number `number` rounded to `precision` decimal places. The method handles the input as a string, in order to avoid rounding errors which would otherwise spoil results beyond the tenth decimal place or so.

**utils.SetDefaultIfNotInArray (value, arr, DefaultIndex)**

Scans the array `arr` for the value specified by the first parameter. `Value` is returned if it exists in the array, otherwise, `arr [DefaultIndex].`

**utils.shl (x, y)**

Bitwise left-shift. Shifts the value `x` left by `y` bits. Equivalent to C++ `<<` operator.

**utils.shr (x, y)**

Bitwise right-shift. Shifts the value `x` right by `y` bits. Equivalent to C++ `>>` operator.

**utils.SortArray (arr, show progress)**

Sorts the array `arr` using a case-insensitive alphabetic sort. Set `show progress` to `True` to see a progress bar while the sort is carried out, or set it to `False` if you don’t want to see a progress bar.

**utils.SortArrayCustom (arr, show progress, plug-in name, method)**

Sorts the array `arr` using a custom sort order routine `method`, which must be passed into this method. `plug-in name` is the name of the plug-in that contains the sort order routine `method`. You can write your own sort order routine: it must be a method that takes two strings (`strA` and `strB`) and returns 1 or 0 based on the results of the comparison.

**utils.SortArrayNumeric (arr, show progress)**

Sorts the array `arr` in ascending numeric order. Set `show progress` to `True` to see a progress bar while the sort is carried out, or set it to `False` if you don’t want to see a progress bar.

**utils.SplitTuplet (tuplet, splitpoint)**

Split the tuplet object `tuplet` at the specified `splitpoint`, which is a number in relation to the tuplet’s parent bar. It then splits a nest of tuplets at that point in the bar.

**utils.StartComponentManager (componentName, callbackFunc)**

Returns an array of filenames (strings) found on the system inside a folder with a given name, following the same rules of precedence as Sibelius’s internal component manager. Files in the user’s application data area take priority over those in the global application data area, followed lastly by those in the Sibelius’s application directory itself.

`callbackFunc` should point to a function in the calling script that scans a supplied directory for files with a specific extension. Such a function might look something like this:

```c
GetFooFiles(dir) { // This is the function signature
    components = CreateArray();
    for each FOO file in dir {
        components[components.NumChildren] = file.NameWithExt;
    }
    return(components);
}
```
Object Reference

In the scenario above, the call to start the component manager would look like this (where "Foo Files" is the name of the directory containing your files):

```csharp
files = utils.StartComponentManager("Foo Files","myPlugin.GetFooFiles");

utils.SubtractFractions(x,y)
Subtracts fraction $y$ from fraction $x$, passed in as ManuScript arrays. Returns an array with the result of the subtraction.

utils.UpperCase(str)
Returns the ANSI string $str$ in uppercase.
Each Score object has a VersionHistory object (obtained by way of the `score.GetVersions()` method), which in turn provides a list of Version objects. Each Version object represents a specific version, and also provides a list of VersionComment objects, which represent the per-version comments (as opposed to bar-attached comments, which are represented to ManuScript as Comment objects, derived from BarObject objects).

**Methods**

**AddVersion** ([*name*, *comment]*)

Adds a new version object and returns it if successful (or null if not), with an optional *name* and *comment* for the version.

**DeleteNthVersion** (*n*)

Deletes the *n*th Version object, returning **True** if successful.

**GetNthVersion** (*n*)

Returns the *n*th Version object.

**Variables**

**NumChildren**

Returns the number of versions in the score’s VersionHistory object.
Object Reference

Version

Accessed via a Score object’s VersionHistory object.

Methods

AddComment (text)

  Adds a new comment with the specified text, and returns the VersionComment object created.

Close()

  Closes all views of the version that are currently open in Sibelius, returning True if it has actually closed anything.

GetNthComment (n)

  Gets the n-th comment as a VersionComment object, or returns null if the index is out of range.

DeleteNthComment (n)

  Deletes the n-th comment, returning True if successful, or null if the index is out of range.

OpenAndReturnScore ()

  Opens the specified version in Sibelius (if it’s not already open) and returns its Score object.

Variables

EndDate

  Returns a DateTime object representing the version’s end date (read only).

IsOpen

  Returns True if the version is currently open in Sibelius (read only).

Name

  Returns the name of the version (read/write).

NumComments

  Returns the number of comments in the version (read only).

StartDate

  Returns a DateTime object representing the version’s start date (read only).
VersionComment

Accessed via Version objects.

Methods
None.

Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text</td>
<td>Returns or changes the text of the comment, and this cannot be undone (read/write).</td>
</tr>
<tr>
<td>TimeStamp</td>
<td>Returns a DateTime object representing the time at which the comment was created.</td>
</tr>
<tr>
<td>UserName</td>
<td>Returns the name of the user who created the comment (read only).</td>
</tr>
</tbody>
</table>
Global constants
Global constants

These are useful variables held internally within ManuScript and are accessible from any plug-in. They are called "constants" because you are encouraged not to change them.

Many of the constants are the names of note values, which you can use to specify a position in a bar easily. So instead of writing 320 you can write Quarter+Sixteenth or equally Crotchet+Semiquaver.

Truth values

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>1</td>
</tr>
<tr>
<td>False</td>
<td>0</td>
</tr>
</tbody>
</table>

Measurements

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Space</td>
<td>32</td>
</tr>
<tr>
<td>StaffHeight</td>
<td>128</td>
</tr>
</tbody>
</table>

Positions and durations

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Long</td>
<td>4096</td>
<td>Sixteenth</td>
</tr>
<tr>
<td>Breve</td>
<td>2048</td>
<td>Semiquaver</td>
</tr>
<tr>
<td>DottedBreve</td>
<td>3072</td>
<td>DottedSixteenth</td>
</tr>
<tr>
<td>Whole</td>
<td>1024</td>
<td>DottedSemiquaver</td>
</tr>
<tr>
<td>Semibreve</td>
<td>1024</td>
<td>ThirtySecond</td>
</tr>
<tr>
<td>DottedWhole</td>
<td>1536</td>
<td>Demisemiquaver</td>
</tr>
<tr>
<td>Half</td>
<td>512</td>
<td>DottedThirtySecond</td>
</tr>
<tr>
<td>Minim</td>
<td>512</td>
<td>DottedDemisemiquaver</td>
</tr>
<tr>
<td>DottedHalf</td>
<td>768</td>
<td>SixtyFourth</td>
</tr>
<tr>
<td>DottedMinim</td>
<td>768</td>
<td>Hemidemisemiquaver</td>
</tr>
<tr>
<td>Quarter</td>
<td>256</td>
<td>DottedSixtyFourth</td>
</tr>
<tr>
<td>Crotchet</td>
<td>256</td>
<td>DottedHemidemisemiquaver</td>
</tr>
<tr>
<td>DottedQuarter</td>
<td>384</td>
<td>OneHundredTwentyEighth</td>
</tr>
<tr>
<td>DottedCrotchet</td>
<td>384</td>
<td>Semihemidemisemiquaver</td>
</tr>
<tr>
<td>Eighth</td>
<td>128</td>
<td>DottedOneHundredTwentyEighth</td>
</tr>
<tr>
<td>Quaver</td>
<td>128</td>
<td>DottedSemihemidemisemiquaver</td>
</tr>
<tr>
<td>Dotted Eighth</td>
<td>192</td>
<td></td>
</tr>
<tr>
<td>DottedQuaver</td>
<td>192</td>
<td></td>
</tr>
</tbody>
</table>

Style names

For the ApplyStyle() method of Score objects. Instead of the capitalized strings in quotes, you can use the equivalent variables in mixed upper and lower case. Note again that the constant HOUSE refers to the options in House Style> Engraving Rules and Layout> Document Setup only; to apply the entire House Style, use the ALLSTYLES constant.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>House</td>
<td>&quot;HOUSE&quot;</td>
<td>Dictionary</td>
</tr>
<tr>
<td>Text</td>
<td>&quot;TEXT&quot;</td>
<td>SpacingRule</td>
</tr>
<tr>
<td>Symbols</td>
<td>&quot;SYMBOLS&quot;</td>
<td>CustomChordNames</td>
</tr>
<tr>
<td>Lines</td>
<td>&quot;LINES&quot;</td>
<td>DefaultPartAppearance</td>
</tr>
<tr>
<td>Noteheads</td>
<td>&quot;NOTEHEADS&quot;</td>
<td>InstrumentsAndEnsembles</td>
</tr>
<tr>
<td>Clefs</td>
<td>&quot;CLEFS&quot;</td>
<td>AllStyles</td>
</tr>
</tbody>
</table>
Global constants

These constants can be used for the format argument of the `AddBarNumber` method.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BarNumberFormatNormal</td>
<td>0</td>
</tr>
<tr>
<td>BarNumberFormatLetterLower</td>
<td>1</td>
</tr>
<tr>
<td>BarNumberFormatLetterUpper</td>
<td>2</td>
</tr>
</tbody>
</table>

Text styles

Here is a list of all the text style identifiers which are guaranteed to be present in any score in Sibelius. In previous versions of Manuscript text styles were identified by a numeric index; this usage has been deprecated but will continue to work for old plug-ins. New plug-ins should use the identifiers given below. For each style we first give the English name of the style and then the identifier.

<table>
<thead>
<tr>
<th>Instrument names</th>
<th>&quot;text.instrumentname&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st and 2nd endings</td>
<td>&quot;text.staff.1st_n_2nd_endings&quot;</td>
</tr>
<tr>
<td>Auto page break warnings</td>
<td>&quot;text.staff.autopagebreak.warnings&quot;</td>
</tr>
<tr>
<td>Boxed text</td>
<td>&quot;text.staff.boxed&quot;</td>
</tr>
<tr>
<td>Expression</td>
<td>&quot;text.staff.expression&quot;</td>
</tr>
<tr>
<td>Chord diagram fingering</td>
<td>&quot;text.staff.fingering.chord_diagrams&quot;</td>
</tr>
<tr>
<td>Footnote</td>
<td>&quot;text.staff.footnote&quot;</td>
</tr>
<tr>
<td>Block lyrics</td>
<td>&quot;text.staff.lyrics.block&quot;</td>
</tr>
<tr>
<td>Multirests (tacet)</td>
<td>&quot;text.staff.multirests.tacet&quot;</td>
</tr>
<tr>
<td>Plain text</td>
<td>&quot;text.staff.plain&quot;</td>
</tr>
<tr>
<td>Small text</td>
<td>&quot;text.staff.small&quot;</td>
</tr>
<tr>
<td>Chord symbol</td>
<td>&quot;text.staff.space.chordsymbol&quot;</td>
</tr>
<tr>
<td>Figured bass</td>
<td>&quot;text.staff.space.figuredbass&quot;</td>
</tr>
<tr>
<td>Fingering</td>
<td>&quot;text.staff.space.fingering&quot;</td>
</tr>
<tr>
<td>Chord diagram fret</td>
<td>&quot;text.staff.space.fretnumbers&quot;</td>
</tr>
<tr>
<td>Lyrics above staff</td>
<td>&quot;text.staff.space.lyrics.above&quot;</td>
</tr>
<tr>
<td>Lyrics (chorus)</td>
<td>&quot;text.staff.space.lyrics.chorus&quot;</td>
</tr>
<tr>
<td>Lyrics line 1</td>
<td>&quot;text.staff.space.lyrics.verse1&quot;</td>
</tr>
<tr>
<td>Lyrics line 2</td>
<td>&quot;text.staff.space.lyrics.verse2&quot;</td>
</tr>
<tr>
<td>Lyrics line 3</td>
<td>&quot;text.staff.space.lyrics.verse3&quot;</td>
</tr>
<tr>
<td>Lyrics line 4</td>
<td>&quot;text.staff.space.lyrics.verse4&quot;</td>
</tr>
<tr>
<td>Lyrics line 5</td>
<td>&quot;text.staff.space.lyrics.verse5&quot;</td>
</tr>
<tr>
<td>Nashville chord numbers</td>
<td>&quot;text.staff.space.nashvillechords&quot;</td>
</tr>
<tr>
<td>Common symbols</td>
<td>&quot;text.staff.symbol.common&quot;</td>
</tr>
<tr>
<td>Figured bass (extras)</td>
<td>&quot;text.staff.symbol.figuredbass.extras&quot;</td>
</tr>
<tr>
<td>Note tails</td>
<td>&quot;text.staff.symbol.notetails&quot;</td>
</tr>
<tr>
<td>Special noteheads etc.</td>
<td>&quot;text.staff.symbol.noteheads.special&quot;</td>
</tr>
<tr>
<td>Percussion instruments</td>
<td>&quot;text.staff.symbol.percussion&quot;</td>
</tr>
<tr>
<td>Special symbols</td>
<td>&quot;text.staff.symbol.special&quot;</td>
</tr>
<tr>
<td>Tablature letters</td>
<td>&quot;text.staff.tab.letters&quot;</td>
</tr>
<tr>
<td>Tablature numbers</td>
<td>&quot;text.staff.tab.numbers&quot;</td>
</tr>
<tr>
<td>Technique</td>
<td>&quot;text.staff.technique&quot;</td>
</tr>
<tr>
<td>Time signatures (one staff only)</td>
<td>&quot;text.staff.timesig.onestaffonly&quot;</td>
</tr>
<tr>
<td>Tuples</td>
<td>&quot;text.staff.tuples&quot;</td>
</tr>
<tr>
<td>Bar numbers</td>
<td>&quot;text.system.barnumber&quot;</td>
</tr>
<tr>
<td>Metronome mark</td>
<td>&quot;text.system.metronome&quot;</td>
</tr>
<tr>
<td>Multirests (numbers)</td>
<td>&quot;text.system.multirestnumbers&quot;</td>
</tr>
<tr>
<td>Composer</td>
<td>&quot;text.system.page_aligned.composer&quot;</td>
</tr>
<tr>
<td>Composer (on title page)</td>
<td>&quot;text.system.page_aligned.composer.ontitlepage&quot;</td>
</tr>
<tr>
<td>Copyright</td>
<td>&quot;text.system.page_aligned.copyright&quot;</td>
</tr>
<tr>
<td>Dedication</td>
<td>&quot;text.system.page_aligned.dedication&quot;</td>
</tr>
<tr>
<td>Footer (inside edge)</td>
<td>&quot;text.system.page_aligned.footer.inside&quot;</td>
</tr>
<tr>
<td>Footer (outside edge)</td>
<td>&quot;text.system.page_aligned.footer.outside&quot;</td>
</tr>
<tr>
<td>Worksheet footer (first page, l)</td>
<td>&quot;text.system.page_aligned.header&quot;</td>
</tr>
<tr>
<td>Header</td>
<td>&quot;text.system.page_aligned.header&quot;</td>
</tr>
<tr>
<td>Worksheet header (first page, l)</td>
<td>&quot;text.system.page_aligned.header_worksheet.left&quot;</td>
</tr>
<tr>
<td>Worksheet header (first page, r)</td>
<td>&quot;text.system.page_aligned.header_worksheet.right&quot;</td>
</tr>
<tr>
<td>Header (after first page)</td>
<td>&quot;text.system.page_aligned.header_notp1&quot;</td>
</tr>
<tr>
<td>Header (after first page, inside edge)</td>
<td>&quot;text.system.page_aligned.header_notp1.inside&quot;</td>
</tr>
<tr>
<td>Instrument name at top left</td>
<td>&quot;text.system.page_aligned.instrumenttopleft&quot;</td>
</tr>
<tr>
<td>Lyricist</td>
<td>&quot;text.system.page_aligned.lyricist&quot;</td>
</tr>
<tr>
<td>Page numbers</td>
<td>&quot;text.system.page_aligned.pagenumber&quot;</td>
</tr>
<tr>
<td>Subtitle</td>
<td>&quot;text.system.page_aligned.subtitle&quot;</td>
</tr>
<tr>
<td>Title (on title page)</td>
<td>&quot;text.system.page_aligned.title.ontitlepage&quot;</td>
</tr>
<tr>
<td>Rehearsal mark</td>
<td>&quot;text.system.rehearsalmarks&quot;</td>
</tr>
<tr>
<td>Repeat (D.C./D.S./To Coda)</td>
<td>&quot;text.system.repeat&quot;</td>
</tr>
<tr>
<td>Tempo</td>
<td>&quot;text.system.tempo&quot;</td>
</tr>
<tr>
<td>Timecode</td>
<td>&quot;text.system.timecode&quot;</td>
</tr>
<tr>
<td>Duration at end of score</td>
<td>&quot;text.system.timecode.duration&quot;</td>
</tr>
<tr>
<td>Hit points</td>
<td>&quot;text.system.timecode.hitpoints&quot;</td>
</tr>
<tr>
<td>Time signatures (huge)</td>
<td>&quot;text.system.timesig.huge&quot;</td>
</tr>
<tr>
<td>Time signatures (large)</td>
<td>&quot;text.system.timesig.large&quot;</td>
</tr>
<tr>
<td>Time signatures</td>
<td>&quot;text.system.timesig.normal&quot;</td>
</tr>
</tbody>
</table>
### Global constants

#### Line styles

<table>
<thead>
<tr>
<th>Style</th>
<th>Code</th>
<th>Style</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arpeggio</td>
<td>line.staff.arpeggio</td>
<td>Dotted slur below</td>
<td>line.staff.slur.down.dotted</td>
</tr>
<tr>
<td>Arpeggio down</td>
<td>line.staff.arpeggio.down</td>
<td>Dashed slur below</td>
<td>line.staff.slur.down.dashed</td>
</tr>
<tr>
<td>Arpeggio up</td>
<td>line.staff.arpeggio.up</td>
<td>Slur above</td>
<td>line.staff.slur.up</td>
</tr>
<tr>
<td>Unused 2</td>
<td>line.staff.arrow</td>
<td>Bracketed slur above</td>
<td>line.staff.slur.up.bracketed</td>
</tr>
<tr>
<td>Arrow</td>
<td>line.staff.arrow.black.right</td>
<td>Dashed slur above</td>
<td>line.staff.slur.up.dashed</td>
</tr>
<tr>
<td>Dashed arrow</td>
<td>line.staff.arrow.blacks.right.dashed</td>
<td>Dotted slur above</td>
<td>line.staff.slur.up.dotted</td>
</tr>
<tr>
<td>Double arrow</td>
<td>line.staff.arrow.blacks.right.left</td>
<td>String indicator above (1)</td>
<td>line.staff.string.above.1</td>
</tr>
<tr>
<td>Vertical arrow (2)</td>
<td>line.staff.arrow.blacks.right.vertical</td>
<td>String indicator above (2)</td>
<td>line.staff.string.above.2</td>
</tr>
<tr>
<td>White arrow</td>
<td>line.staff.arrow.white.right</td>
<td>String indicator above (3)</td>
<td>line.staff.string.above.3</td>
</tr>
<tr>
<td>Dashed white arrow</td>
<td>line.staff.arrow.white.right.dashed</td>
<td>String indicator above (4)</td>
<td>line.staff.string.above.4</td>
</tr>
<tr>
<td>Double white arrow</td>
<td>line.staff.arrow.white.right.left</td>
<td>String indicator above (5)</td>
<td>line.staff.string.above.5</td>
</tr>
<tr>
<td>Vertical arrow</td>
<td>line.staff.arrow.white.vertical</td>
<td>String indicator above (6)</td>
<td>line.staff.string.above.6</td>
</tr>
<tr>
<td>Beam</td>
<td>line.staff.beam</td>
<td>String indicator above (7)</td>
<td>line.staff.string.above.7</td>
</tr>
<tr>
<td>Guitar Bend</td>
<td>line.staff.bend</td>
<td>String indicator above (8)</td>
<td>line.staff.string.above.8</td>
</tr>
<tr>
<td>Guitar hold bend</td>
<td>line.staff.bend.hold</td>
<td>String indicator below (1)</td>
<td>line.staff.string.below.1</td>
</tr>
<tr>
<td>Box</td>
<td>line.staff.box</td>
<td>String indicator below (2)</td>
<td>line.staff.string.below.2</td>
</tr>
<tr>
<td>Bracket above</td>
<td>line.staff.bracket.above</td>
<td>String indicator below (3)</td>
<td>line.staff.string.below.3</td>
</tr>
<tr>
<td>Bracket above (end)</td>
<td>line.staff.bracket.above.end</td>
<td>String indicator below (4)</td>
<td>line.staff.string.below.4</td>
</tr>
<tr>
<td>Bracket above (start)</td>
<td>line.staff.bracket.above.start</td>
<td>String indicator below (5)</td>
<td>line.staff.string.below.5</td>
</tr>
<tr>
<td>Bracket below</td>
<td>line.staff.bracket.below</td>
<td>String indicator below (6)</td>
<td>line.staff.string.below.6</td>
</tr>
<tr>
<td>Bracket below (end)</td>
<td>line.staff.bracket.below.end</td>
<td>String indicator below (7)</td>
<td>line.staff.string.below.7</td>
</tr>
<tr>
<td>Bracket below (start)</td>
<td>line.staff.bracket.below.start</td>
<td>String indicator below (8)</td>
<td>line.staff.string.below.8</td>
</tr>
<tr>
<td>Vertical bracket</td>
<td>line.staff.bracket.vertical</td>
<td>Tie</td>
<td>line.staff.tie</td>
</tr>
<tr>
<td>Vertical bracket 2</td>
<td>line.staff.bracket.vertical.2</td>
<td>Trill</td>
<td>line.staff.trill</td>
</tr>
<tr>
<td>Dashed line</td>
<td>line.staff.dashed</td>
<td>Tuplet</td>
<td>line.staff.tuplet</td>
</tr>
<tr>
<td>Dotted line</td>
<td>line.staff.dotted</td>
<td>Vertical line</td>
<td>line.staff.vertical</td>
</tr>
<tr>
<td>Glissando (straight)</td>
<td>line.staff.gliss.straight</td>
<td>Vibrato</td>
<td>line.staff.vibrato.bar</td>
</tr>
<tr>
<td>Glissando (wavy)</td>
<td>line.staff.gliss.wavy</td>
<td>Wide vibrato</td>
<td>line.staff.vibrato.wide</td>
</tr>
<tr>
<td>Guitar effect</td>
<td>line.staff.guitareffect</td>
<td>Dashed system line</td>
<td>line.system.dashed</td>
</tr>
<tr>
<td>Crescendo</td>
<td>line.staff.hairpin.crescendo</td>
<td>Wide dashed system line</td>
<td>line.system.dashed.wide</td>
</tr>
<tr>
<td>Bracketed crescendo</td>
<td>line.staff.hairpin.crescendo.bracketed</td>
<td>1st ending</td>
<td>line.system.repeat.1st</td>
</tr>
<tr>
<td>Dashed crescendo</td>
<td>line.staff.hairpin.crescendo.dashed</td>
<td>1st and 2nd ending</td>
<td>line.system.repeat.1st_n_2nd</td>
</tr>
<tr>
<td>Dotted crescendo</td>
<td>line.staff.hairpin.crescendo.dotted</td>
<td>2nd ending</td>
<td>line.system.repeat.2nd</td>
</tr>
<tr>
<td>Crescendo from silence</td>
<td>line.staff.hairpin.crescendo.fromsilence</td>
<td>2nd ending (closed)</td>
<td>line.system.repeat.2nd.closed</td>
</tr>
<tr>
<td>Diminuendo</td>
<td>line.staff.hairpin.diminuendo</td>
<td>3rd ending</td>
<td>line.system.repeat.3rd</td>
</tr>
<tr>
<td>Bracketed diminuendo</td>
<td>line.staff.hairpin.diminuendo.bracketed</td>
<td>Repeat ending (closed)</td>
<td>line.system.repeat.closed</td>
</tr>
<tr>
<td>Dashed diminuendo</td>
<td>line.staff.hairpin.diminuendo.dashed</td>
<td>Repeat ending (open)</td>
<td>line.system.repeat.open</td>
</tr>
<tr>
<td>Dotted diminuendo</td>
<td>line.staff.hairpin.diminuendo.dotted</td>
<td>Accel.</td>
<td>line.system.tempo.accel</td>
</tr>
<tr>
<td>Diminuendo to silence</td>
<td>line.staff.hairpin.diminuendo.tosilence</td>
<td>Accel. (italic)</td>
<td>line.system.tempo.accel.italic</td>
</tr>
<tr>
<td>Guitar artificial harmonic</td>
<td>line.staff.harmonic.artificial</td>
<td>Accel. (italic, text only)</td>
<td>line.system.tempo.accel.italic.textonly</td>
</tr>
<tr>
<td>Guitar harp harmonic</td>
<td>line.staff.harmonic.harp</td>
<td>Molto accel.</td>
<td>line.system.tempo.accel.molto</td>
</tr>
<tr>
<td>Guitar pinch harmonic</td>
<td>line.staff.harmonic.pinch</td>
<td>Molto accel. (text only)</td>
<td>line.system.tempo.accel.molto.textonly</td>
</tr>
<tr>
<td>Guitar touch harmonic</td>
<td>line.staff.harmonic.touch</td>
<td>Poco accel.</td>
<td>line.system.tempo.accel.poco</td>
</tr>
<tr>
<td>Guitar harmonics</td>
<td>line.staff.harmonics</td>
<td>Poco accel. (text only)</td>
<td>line.system.tempo.accel.poco.textonly</td>
</tr>
<tr>
<td>Hauptstimme</td>
<td>line.staff.hauptstimme</td>
<td>Accel. (text only)</td>
<td>line.system.tempo.accel.textonly</td>
</tr>
<tr>
<td>Guitar let ring</td>
<td>line.staff.letring</td>
<td>Tempo change (arrow right)</td>
<td>line.system.tempo.arrowright</td>
</tr>
<tr>
<td>Lyric line</td>
<td>line.staff.lyric</td>
<td>Rall.</td>
<td>line.system.tempo.rall</td>
</tr>
<tr>
<td>Guitar palm mute</td>
<td>line.staff.mute.palm</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

138
Global constants

Clef styles

Here is a list of all the clef style identifiers that are guaranteed to be present in any score in Sibelius, for use with the `Stave.AddClef` method. For each style we first give the English name of the style, and then the identifier.

<table>
<thead>
<tr>
<th>Style</th>
<th>Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alto</td>
<td><code>clef.alto</code></td>
</tr>
<tr>
<td>Baritone C</td>
<td><code>clef.baritone.c</code></td>
</tr>
<tr>
<td>Baritone F</td>
<td><code>clef.baritone.f</code></td>
</tr>
<tr>
<td>Bass</td>
<td><code>clef.bass</code></td>
</tr>
<tr>
<td>Bass down 8</td>
<td><code>clef.bass.down.8</code></td>
</tr>
<tr>
<td>Bass up 15</td>
<td><code>clef.bass.up.15</code></td>
</tr>
<tr>
<td>Bass up 8</td>
<td><code>clef.bass.up.8</code></td>
</tr>
<tr>
<td>Null</td>
<td><code>clef.null</code></td>
</tr>
<tr>
<td>Percussion</td>
<td><code>clef.percussion</code></td>
</tr>
<tr>
<td>Percussion 2</td>
<td><code>clef.percussion_2</code></td>
</tr>
<tr>
<td>Soprano</td>
<td><code>clef.soprano</code></td>
</tr>
<tr>
<td>Mezzo-soprano</td>
<td><code>clef.soprano.mezzo</code></td>
</tr>
<tr>
<td>Tab</td>
<td><code>clef.tab</code></td>
</tr>
<tr>
<td>Small tab</td>
<td><code>clef.tab.small</code></td>
</tr>
<tr>
<td>Small tab (taller)</td>
<td><code>clef.tab.small.taller</code></td>
</tr>
<tr>
<td>Tab (taller)</td>
<td><code>clef.tab.taller</code></td>
</tr>
<tr>
<td>Tenor</td>
<td><code>clef.tenor</code></td>
</tr>
<tr>
<td>Tenor down 8</td>
<td><code>clef.tenor.down.8</code></td>
</tr>
<tr>
<td>Treble</td>
<td><code>clef.treble</code></td>
</tr>
<tr>
<td>Treble down 8</td>
<td><code>clef.treble.down.8</code></td>
</tr>
<tr>
<td>Treble (down 8)</td>
<td><code>clef.treble.down.8.bracketed</code></td>
</tr>
<tr>
<td>Treble up 15</td>
<td><code>clef.treble.up.15</code></td>
</tr>
<tr>
<td>Treble up 8</td>
<td><code>clef.treble.up.8</code></td>
</tr>
<tr>
<td>French violin</td>
<td><code>clef.violin.french</code></td>
</tr>
<tr>
<td>Sub-bass F</td>
<td><code>clef.sub-bass.f</code></td>
</tr>
</tbody>
</table>

Instrument types

Here is a list of all the instrument type identifiers that are guaranteed to be present in any score in Sibelius. For each style we first give the English name of the style and then the identifier. Note that only the tablature stave types can be used with guitar frames; the rest are included for completeness.

<table>
<thead>
<tr>
<th>Instrument</th>
<th>Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alp-Horn in F</td>
<td>instrument.brass.alp-horn.f</td>
</tr>
<tr>
<td>Alp-Horn in G</td>
<td>instrument.brass.alp-horn.g</td>
</tr>
<tr>
<td>Baritone Bugle in G</td>
<td>instrument.brass.bugle.baritone.g</td>
</tr>
<tr>
<td>Contrabass Bugle in G</td>
<td>instrument.brass.bugle.contrabass.g</td>
</tr>
<tr>
<td>Euphonium Bugle in G</td>
<td>instrument.brass.bugle.euphonium.g</td>
</tr>
<tr>
<td>Mellophone Bugle in G</td>
<td>instrument.brass.bugle.mellophone.g</td>
</tr>
<tr>
<td>Soprano Bugle in G</td>
<td>instrument.brass.bugle.soprano.g</td>
</tr>
<tr>
<td>Cimbasso in Bb</td>
<td>instrument.brass.cimbasso.bflat</td>
</tr>
<tr>
<td>Cimbasso in Eb</td>
<td>instrument.brass.cimbasso.eflat</td>
</tr>
<tr>
<td>Cimbasso in F</td>
<td>instrument.brass.cimbasso.f</td>
</tr>
<tr>
<td>Cornet in A</td>
<td>instrument.brass.cornet.a</td>
</tr>
<tr>
<td>Cornet in Bb</td>
<td>instrument.brass.cornet.bflat</td>
</tr>
</tbody>
</table>
Global constants

Soprano Cornet in Eb
Euphonium in Bb [treble clef]
Euphonium in Bb [bass clef, treble transp.]
Euphonium in C [bass clef]
Euphonium in Bb [bass clef]
Flugelhorn
Horn in A [no key]
Horn in Ab alto [no key]
Alto Horn in Eb
Alto Horn in F
Horn in B [no key]
Baritone in Bb [treble clef]
Baritone in C [treble clef]
Baritone in Bb [bass clef, treble transp.]
Baritone in C [bass clef]
Bass in Bb
Bass in Bb [bass clef, treble transp.]
Bass in C
Bass in Eb
Bass in Eb [bass clef, treble transp.]
A Basso Horn [no key]
Bb Basso Horn [no key]
C Basso Horn [no key]
Horn in Bb [no key]
Horn in C [no key]
Horn in D [no key]
Horn in Db [no key]
Horn in E [no key]
Horn in Eb
Horn in Eb [no key]
Horn in F
Horn in F [bass clef]
Horn in F# [no key]
Horn in G [no key]
Tenor Horn
Mellophone in Eb
Mellophone in F
Mellophonium in Eb
Mellophonium in F
Ophicleide
Brass
Serpent
Sousaphone in Bb
Sousaphone in Eb
Trombone
Alto Trombone
Bass Trombone
Trombone in Bb [bass clef, treble transp.]
Contrabass Trombone

instrument.brass.cornet.soprano.eflat
instrument.brass.euphonium
instrument.brass.euphonium.bassclef
instrument.brass.euphonium.bflat.bassclef.bassclef
instrument.brass.flugelhorn
instrument.brass.horn.a.nokeysig
instrument.brass.horn.alto.aflat.nokeysig
instrument.brass.horn.alto.eflat
instrument.brass.horn.alto.f
instrument.brass.horn.b.nokeysig
instrument.brass.horn.baritone
instrument.brass.horn.baritone.2
instrument.brass.horn.baritone.bassclef
instrument.brass.horn.baritone.bassclef.bassclef
instrument.brass.horn.bass.bflat
instrument.brass.horn.bass.bflat.bassclef
instrument.brass.horn.bass.c
instrument.brass.horn.bass.eflat
instrument.brass.horn.bass.eflat.bassclef
instrument.brass.horn.basso.a.nokeysig
instrument.brass.horn.basso.bflat.nokeysig
instrument.brass.horn.basso.c.nokeysig
instrument.brass.horn.bflat.nokeysig
instrument.brass.horn.c.nokeysig
instrument.brass.horn.d.nokeysig
instrument.brass.horn.db.nokeysig
instrument.brass.horn.e.nokeysig
instrument.brass.horn.e.flat
instrument.brass.horn.e.flat.nokeysig
instrument.brass.horn.f
instrument.brass.horn.f.bassclef
instrument.brass.horn.f.nokeysig
instrument.brass.horn.fsharp.nokeysig
instrument.brass.horn.g.nokeysig
instrument.brass.horn.tenor
instrument.brass.mellophone.eflat
instrument.brass.mellophone.f
instrument.brass.mellophonium.eflat
instrument.brass.mellophonium.f
instrument.brass.ophicleide
instrument.brass.section
instrument.brass.serpent
instrument.brass.sousaphone.bflat
instrument.brass.sousaphone.eflat
instrument.brass.trombone
instrument.brass.trombone.alto
instrument.brass.trombone.bass
instrument.brass.trombone.bassclef.trebleclef
instrument.brass.trombone.contrabass
Global constants

Tenor Trombone
Trombone in Bb [treble clef]
Trumpet in A
Trumpet in B [no key]
Bass Trumpet in Bb
Bass Trumpet in Eb
Trumpet in Bb
Trumpet in Bb [no key]
Trumpet in C
Trumpet in D
Trumpet in Db
Trumpet in E [no key]
Trumpet in Eb
Trumpet in F
Trumpet in G [no key]
Piccolo Trumpet in A
Piccolo Trumpet in Bb
Tenor Trumpet in Eb
Tuba
Tuba in F
Tenor Tuba (Wagner, in Bb)
Tenor Tuba [bass clef]
Wagner Tuba in Bb
Wagner Tuba in F
Applause
Birdsong
Helicopter
Ondes Martenot
Sampler
Seashore
Tape
Telephone
Theremin
Bajo [notation]
Bajo, 6-string [tab]
Bajo, 4-string [tab]
Bajo, 5-string [tab]
Alto Balalaika [notation]
Alto Balalaika [tab]
Bass Balalaika [notation]
Bass Balalaika [tab]
Contrabass Balalaika [notation]
Contrabass Balalaika [tab]
Prima Balalaika [notation]
Prima Balalaika [tab]
Second Balalaika [notation]
Second Balalaika [tab]
Bandola [notation]
Bandola [tab]
Bandolón [notation]
Global constants

Bandolón [tab]  instrument.fretted.bandolon.tab
Bandurria [tab]  instrument.fretted.bandurria.tab
Banjo [tab]  instrument.fretted.banjo.tab
Banjo (aDADE tuning) [tab]  instrument.fretted.banjo.aDADE.tab
Banjo (aEADE tuning) [tab]  instrument.fretted.banjo.aEADE.tab
Banjo (gCGRD tuning) [tab]  instrument.fretted.banjo.gCGRD.tab
Banjo (gCGCD tuning) [tab]  instrument.fretted.banjo.gCGCD.tab
Banjo (gDFAD tuning) [tab]  instrument.fretted.banjo.gDFAD.tab
Banjo (gDGBD tuning) [tab]  instrument.fretted.banjo.gDGBD.tab
Banjo (gDGCD tuning) [tab]  instrument.fretted.banjo.gDGCD.tab
Tenor Banjo [tab]  instrument.fretted.banjo.tenor.tab
Bordonúa [tab]  instrument.fretted.bordonua.tab
Cavaquinho [tab]  instrument.fretted.cavaquinho.tab
Charango [tab]  instrument.fretted.charango.tab
Cuatro [tab]  instrument.fretted.cuatro.tab
Cavaquinho (DAA tuning) [tab]  instrument.fretted.cavaquinho.daa.tab
Cavaquinho (DAD tuning) [tab]  instrument.fretted.cavaquinho.dad.tab
Gamba [tab]  instrument.fretted.gamba.tab
Dulcimer [tab]  instrument.fretted.dulcimer.tab
Dulcimer (DAD tuning) [tab]  instrument.fretted.dulcimer.dad.tab
12-string Acoustic Guitar [tab]  instrument.fretted.guitar.12-string.tab
12-string Acoustic Guitar, DADGAD tuning [tab]  instrument.fretted.guitar.12-string.dadgad.tab
12-string Acoustic Guitar, DAD tuning [tab]  instrument.fretted.guitar.12-string.dad.tab
12-string Acoustic Guitar, dropped D tuning [tab]  instrument.fretted.guitar.12-string.dropped-d.tab
12-string Acoustic Guitar, open D tuning [tab]  instrument.fretted.guitar.12-string.open-d.tab
12-string Acoustic Guitar, open E tuning [tab]  instrument.fretted.guitar.12-string.open-e.tab
12-string Acoustic Guitar, open G tuning [tab]  instrument.fretted.guitar.12-string.open-g.tab
12-string Acoustic Guitar, standard tuning (no rhythms) [tab]  instrument.fretted.guitar.12-string.tab.rhythms
12-string Acoustic Guitar, standard tuning [tab]  instrument.fretted.guitar.12-string.tab
Acoustic Guitar [tab]  instrument.fretted.guitar.acoustic.tab
Acoustic Guitar, DADGAD tuning [tab]
Acoustic Guitar, double D tuning [tab]
Acoustic Guitar, modal D tuning [tab]
Acoustic Guitar, Nashville tuning [tab]
Acoustic Guitar, open A tuning [tab]
Acoustic Guitar, open C tuning [tab]
Acoustic Guitar, open D tuning [tab]
Acoustic Guitar, open Dm cross-note tuning [tab]
Acoustic Guitar, open E tuning [tab]
Acoustic Guitar, open G tuning [tab]
Acoustic Guitar, standard tuning (no rhythms) [tab]
Acoustic Bass [notation]
Acoustic Bass [tab]
4-string Bass Guitar [notation]
4-string Bass Guitar [tab]
5-string Bass Guitar [notation]
5-string Bass Guitar [tab]
Bass Guitar [notation]
6-string Bass Guitar [notation]
6-string Bass Guitar [tab]
Acoustic Bass [notation]
5-string Electric Bass [notation]
5-string Electric Bass [tab]
Electric Bass [notation]
6-string Electric Bass [notation]
6-string Electric Bass [tab]
Fretless Electric Bass [tab]
Fretless Electric Bass [notation]
6-string Fretless Electric Bass [notation]
6-string Fretless Electric Bass [tab]
Fretless Bass Guitar [tab]
Semi-Acoustic Bass [notation]
Semi-Acoustic Bass [tab]
Bass Guitar [tab]
Bass Guitar [tab, with rhythms]
Classical Guitar [notation]
Classical Guitar, DADGAD tuning [tab]
Classical Guitar, double D tuning [tab]
Classical Guitar, dropped D tuning [tab]
Classical Guitar, open D tuning [tab]
Classical Guitar, open E tuning [tab]
Global constants

Classical Guitar, open G tuning [tab]
Classical Guitar, standard tuning (no rhythms) [tab]
Classical Guitar, standard tuning [tab]
Electric Guitar [notation]
7-string Electric Guitar, low A tuning [tab]
7-string Electric Guitar, low B tuning [tab]
Electric Guitar, DADGAD tuning [tab]
Electric Guitar, double D tuning [tab]
Electric Guitar, dropped D tuning [tab]
Electric Guitar, open D tuning [tab]
Electric Guitar, open E tuning [tab]
Electric Guitar, open G tuning [tab]
Electric Guitar, standard tuning (no rhythms) [tab]
Electric Guitar, standard tuning [tab]
Kora
Semi-acoustic Guitar [notation]
Semi-acoustic Guitar, DADGAD tuning [tab]
Semi-acoustic Guitar, double D tuning [tab]
Semi-acoustic Guitar, dropped D tuning [tab]
Semi-acoustic Guitar, open D tuning [tab]
Semi-acoustic Guitar, open E tuning [tab]
Semi-acoustic Guitar, open G tuning [tab]
Semi-acoustic Guitar, standard tuning (no rhythms) [tab]
Semi-acoustic Guitar, standard tuning [tab]
10-string Hawaiian Steel Guitar [tab]
Hawaiian Steel Guitar [notation]
6-string Hawaiian Steel Guitar, standard tuning [tab]
6-string Hawaiian Steel Guitar, alternate tuning [tab]
6-string Hawaiian Steel Guitar, slack key Bb Mauna Loa tuning [tab]
6-string Hawaiian Steel Guitar, slack key C Mauna Loa tuning [tab]
6-string Hawaiian Steel Guitar, slack key Wahine CGDGDG tuning [tab]
6-string Hawaiian Steel Guitar, slack key Wahine CGDGBE tuning [tab]
6-string Hawaiian Steel Guitar, slack key Wahine DGDF#BD tuning [tab]
6-string Hawaiian Steel Guitar, slack key G Mauna Loa tuning [tab]
6-string Hawaiian Steel Guitar, slack key G Taro Patch tuning [tab]
6-string Hawaiian Steel Guitar, slack key Wahine GCDGBE tuning [tab]
8-string Hawaiian Steel Guitar [tab]
8-string Hawaiian Steel Guitar, alternate tuning [tab]
Hawaiian Steel Guitar [tab]
Pedal Steel Guitar [notation]
Pedal Steel Guitar [tab]
Guitarra [notation]
Guitarra, Coimbra [tab]
Guitarra, Lisboa [tab]
Guitarra, Portuguesa [tab]
Guitarrón [notation]
Guitarrón [tab]
Láud [notation]
Láud [tab]
Tenor Lute [notation]
Global constants

Bass Lute [notation]
Bass Lute, D tuning, French/English [tab]
Bass Lute, D tuning, Italian [tab]
Bass Lute, D tuning, Spanish [tab]
Tenor Lute, G tuning, Italian [tab]
Tenor Lute, G tuning, Spanish [tab]
Tenor Lute, G tuning, French/English [tab]
Tenor Lute, A tuning, French/English [tab]
Tenor Lute, A tuning, Italian [tab]
Tenor Lute, A tuning, Spanish [tab]
Treble Lute [notation]
Treble Lute, D tuning, French/English [tab]
Treble Lute, D tuning, Italian [tab]
Treble Lute, D tuning, Spanish [tab]
Mandolin [notation]
Oud [notation]
Oud [tab]
Qanoun
Requinto [notation]
Requinto [tab]
Santoor
Sitar [notation]
Sitar (Ravi Shankar) [tab]
Sitar (Vilayat Khan) [tab]
Tambura (Female) [notation]
Tambura (Male) [notation]
Tiple [notation]
Tiple, Argentina [tab]
Tiple, Colombia ADF#B tuning [tab]
Tiple, Colombia DGBE tuning [tab]
Tiple, Cuba [tab]
Tiple, Peru [tab]
Tiple, Santo Domingo [tab]
Tiple, Uruguay [tab]
Tres [notation]
Tres, GCE tuning [tab]
Tres, ADF# tuning [tab]
Tres, GBE tuning [tab]
Ukulele [notation]
Ukulele [tab]
Vihuela [notation]
Vihuela [tab]
Zither
Keyboard
Accordion
Bandeoneon
Celesta
Clavichord
Harmonium
Global constants

- Harpsichord
- Keyboards
- Tape Sampler Keyboard [Brass]
- Tape Sampler Keyboard [Choir]
- Tape Sampler Keyboard [Flute]
- Tape Sampler Keyboard [Strings]
- Melodeon
- Electric Organ
- Organ [manuals]
- Manual [solo organ manuals]
- Ped. [Organ pedals]
- Pedal [solo organ pedals]
- Piano
- Electric Piano
- Electric Clavichord
- Electric Stage Piano
- Overdriven Electric Piano
- Honky-tonk Piano
- Synthesizer
- Unnamed (2 lines)
- Unnamed (3 lines)
- Unnamed (4 lines)
- Unnamed (bass staff)
- No instrument (barlines shown)
- No instrument (bar rests shown)
- No instrument (hidden)
- Solo
- Unnamed (treble staff)
- Almglocken
- Antique Cymbals
- Chimes
- Chimes [no key]
- Bell lyre [marching band]
- Orchestral Bells
- Tubular Bells
- Cimbalom
- Crotales
- Steel Drums
- Steel Drums [bass clef, treble transp.]
- Gamelan Kengong
- Gamelan Slentam
- Glockenspiel
- Alto Glockenspiel
- Soprano Glockenspiel
- Handbells
- Harp
- Lever Harp
- Kalimba
- Marimba [grand staff]
- Marimba [treble staff]

instrument.keyboard.harpsichord
instrument.keyboard.keyboards
instrument.keyboard.tape sampler.brass
instrument.keyboard.tape sampler.choir
instrument.keyboard.tape sampler.flute
instrument.keyboard.tape sampler
instrument.keyboard.melodeon
instrument.keyboard.organ.electric
instrument.keyboard.organ.manuals
instrument.keyboard.organ.manuals.solo
instrument.keyboard.organ.pedals
instrument.keyboard.organ.pedals.solo
instrument.keyboard.piano
instrument.keyboard.piano.electric
instrument.keyboard.piano.electric.clavichord
instrument.keyboard.piano.electric.stage
instrument.keyboard.piano.electric.overdriven
instrument.keyboard.piano.honky-tonk
instrument.keyboard.synthesizer
instrument.other.2lines
instrument.other.3lines
instrument.other.4lines
instrument.other.bassclef
instrument.other.none.barlines
instrument.other.none.barrests
instrument.other.none.hidden
instrument.other.solo.real
instrument.other.trebleclef
instrument.pitchedpercussion.almglocken
instrument.pitchedpercussion.antiquecymbals
instrument.pitchedpercussion.bells.chimes
instrument.pitchedpercussion.bells.chimes.nokeysig
instrument.pitchedpercussion.bells.marching
instrument.pitchedpercussion.bells.orchestral
instrument.pitchedpercussion.bells.tubular
instrument.pitchedpercussion.cimbalom
instrument.pitchedpercussion.crotales
instrument.pitchedpercussion.drums.steel
instrument.pitchedpercussion.drums.steel.bassclef
instrument.pitchedpercussion.gamelan.kengong
instrument.pitchedpercussion.gamelan.slentam
instrument.pitchedpercussion.glockenspiel
instrument.pitchedpercussion.glockenspiel.alto
instrument.pitchedpercussion.glockenspiel.soprano
instrument.pitchedpercussion.handbells
instrument.pitchedpercussion.harp
instrument.pitchedpercussion.harp.lever
instrument.pitchedpercussion.kalimba
instrument.pitchedpercussion.marimba
instrument.pitchedpercussion.marimba.trebleclef
Global constants

Alto Metallophone instrument.pitchedpercussion.metallophone.alto
Bass Metallophone instrument.pitchedpercussion.metallophone.bass
Soprano Metallophone instrument.pitchedpercussion.metallophone.soprano
Roto-toms instrument.pitchedpercussion.roto-toms
Temple Blocks instrument.pitchedpercussion.templeblocks
timpani instrument.pitchedpercussion.timpani
Timpani [with key] instrument.pitchedpercussion.timpani
Timpani [no key] instrument.pitchedpercussion.timpani.nokeysig
Vibraphone instrument.pitchedpercussion.vibraphone
Wood Blocks [5 lines] instrument.pitchedpercussion.woodblocks
Xylophone instrument.pitchedpercussion.xylophone
Alto Xylophone instrument.pitchedpercussion.xylophone.alto
Bass Xylophone instrument.pitchedpercussion.xylophone.bass
Contra Bass Bar instrument.pitchedpercussion.xylophone.contrabass.bar
Gyil instrument.pitchedpercussion.xylophone.gyil
Soprano Xylophone instrument.pitchedpercussion.xylophone.soprano
Xylorimba instrument.pitchedpercussion.xylorimba
Alto instrument.singers.alto
Solo Alto instrument.singers.alto.solo
Altus instrument.singers.altus
Baritone instrument.singers.baritone
Solo Baritone instrument.singers.baritone.solo
Bass instrument.singers.bass
Solo Bass instrument.singers.bass.solo
Bassus instrument.singers.bassus
Cantus instrument.singers.cantus
Choir instrument.singers.choir
Contralto instrument.singers.contralto
Countertenor instrument.singers.counter-tenor
Mean instrument.singers.mean
Mezzo-soprano instrument.singers.mezzo-soprano
Quintus instrument.singers.quintus
Secundus instrument.singers.secundus
Soprano instrument.singers.soprano
Solo Soprano instrument.singers.soprano.solo
Tenor instrument.singers.tenor
Solo Tenor instrument.singers.tenor.solo
Treble instrument.singers.treble
Solo Treble instrument.singers.treble.solo
Voice instrument.singers.voice
Voice [male] instrument.singers.voice.male
Contrabass instrument.strings.contrabass
Bass [Double] instrument.strings.contrabass.bass
Double Bass instrument.strings.contrabass.double-bass
Solo Contrabass instrument.strings.contrabass.solo
String Bass instrument.strings.contrabass.string
Upright Bass instrument.strings.contrabass.upright
Hurdy-gurdy instrument.strings.hurdy-gurdy
Sarangi instrument.strings.sarangi
Strings instrument.strings.section
Strings [reduction] instrument.strings.section.reduction
Global constants

Bass Viol
Tenor Viol
Treble Viol
Viola
Solo Viola
Violin 1
Violin 2
Violin I
Violin II
Solo Violin
Violoncello
Solo Violoncello
Anvil
Cha-cha bell [1 line]
Mambo bell [1 line]
Sleigh Bells
Brake Drum [1 line]
Cabasa [1 line]
Cabasa [2 lines]
Castanets
Shaker, Caxixi [1 line]
Claves [1 line]
Shaker, Cocoa Bean Rattle [1 line]
Finger Cymbals [1 line]
Percussion [1 line]
Percussion [2 lines]
Berimbau
Percussion [3 lines]
Percussion [4 lines]
Percussion [5 lines]
Agogos [2 lines]
Bass Drum
Marching Bass Drum [3 lines]
Marching Bass Drum [5 lines]
Itótele [Batá Drum]
Iyá [Batá Drum]
Okónkolo [Batá Drum]
Bongos [2 lines]
Bongo Bell [High]
Bongo Bell [Low]
Box
Cajon [2 lines]
Congas [2 lines]
Congas [1 line]
Congas [3 lines]
Congas [4 lines]
Cuica [3 lines]
Cymbals
Marching Cymbals [5 lines]
Djembe [3 lines]
Drum Set (Rock)
Drum Set (Alternative)
Drum Set (Brushes)
Drum Set (Dance)
Drum Set (Disco)
Drum Set (Electronica)
Drum Set (Fusion)
Drum Set (Garage)
Drum Set (Hip-hop)
Drum Set (Industrial)
Drum Set (Jazz)
Drum Set (Lo-Fi)
Drum Set (Metal)
Drum Set (Motown)
Drum Set (New Age)
Drum Set (Pop)
Drum Set (Reggae)
Drum Set (Stadium Rock)
Drum Set (Rods)
Drum Set (Drum Machine)
Dumbek [3 lines]
Kidi [Ewe Drum]
Sogo [Ewe Drum]
Ganko Kwe (Bell)
Jam Blocks [2 lines]
Jawbone [1 line]
Pandeiro [2 lines]
Rain Stick (High) [1 line]
Rain Stick (Low) [1 line]
Egg Shaker (High) [1 line]
Egg Shaker (Low) [1 line]
Egg Shaker (Medium) [1 line]
Side Drum
Snare Drum
Marching Snare Drums [5 lines]
Surdo [2 lines]
Tabla
Taiko Drum
Tenor Drum
Marching Tenor Drums [5 lines]
Quads [5 lines]
Tom-toms [5 lines]
Tom-toms [4 lines]
Udu
Shaker, Egg Shaker [1 line]
Finger Click [1 line]
Gamelan Gong Ageng (High) [1 line]
Gamelan Gong Ageng (Low) [1 line]
Gamelan Kempyang and Ketuk [2 lines]
Global constants

Gamelan Khendang Ageng [1 line]
Gamelan Khendang Ciblon [1 line]
Large Gong [1 line]
Medium Gong [1 line]
Gourd [1 line]
Guira [1 line]
Guiro (High) [1 line]
Guiro (Medium) [1 line]
Handclap [1 line]
Shaker, Kayamba [1 line]
Maracas
Shaker, Gourd Maracas [1 line]
Maracas (High)
Maracas (Medium)
Mark tree [1 line]
Shaker, Nsak Rattle [1 line]
Finger Snaps
Hand Claps
Patsch
Stamp
Salsa bell [1 line]
Shaker [1 line]
Shaker, Shekere [1 line]
Tam-tam
Tambourine
Timbales [2 lines]
Timbales [5 lines]
Triangle
Shaker, Wasembe Rattle (High) [1 line]
Shaker, Wasembe Rattle (Low) [1 line]
Shaker, Wasembe Rattle (Medium) [1 line]
Whip
Whistle
Wind Chimes [1 line]
Wood Block [1 line]
Bagpipes
Basset Horn
Bassoon
Contrabassoon
Quart Bassoon
Quint Bassoon
Clarinet in A
Clarinet in Ab
Alto Clarinet in Eb
Alto Clarinet in Eb [bass clef, treble transp.]
Bass Clarinet in Bb
Bass Clarinet in Bb [score sounds 8vb]
Bass Clarinet in Bb [bass clef, treble transp.]
Clarinet in Bb
Clarinet in C

instrument.unpitched.gamelan.khendang-ageng
instrument.unpitched.gamelan.khendang-ciblon
instrument.unpitched.gong.large.1line
instrument.unpitched.gong.medium.1line
instrument.unpitched.gourd
instrument.unpitched.guira
instrument.unpitched.guiro.high
instrument.unpitched.guiro.medium
instrument.unpitched.handclap
instrument.unpitched.kayamba.1line
instrument.unpitched.maracas
instrument.unpitched.maracas.gourd.1line
instrument.unpitched.maracas.high
instrument.unpitched.maracas.medium
instrument.unpitched.marktree
instrument.unpitched.nsk.1line
instrument.unpitched.orff.fingersnaps
instrument.unpitched.orff.handclaps
instrument.unpitched.orff.patsch
instrument.unpitched.orff.stamp
instrument.unpitched.salsa.bell
instrument.unpitched.shaker
instrument.unpitched.shakere.1line
instrument.unpitched.tam-tam
instrument.unpitched.tambourine
instrument.unpitched.timbales.2lines
instrument.unpitched.timbales.5lines
instrument.unpitched.triangle
instrument.unpitched.wasembe.high.1line
instrument.unpitched.wasembe.low.1line
instrument.unpitched.wasembe.medium.1line
instrument.unpitched.whip
instrument.unpitched.whistle
instrument.unpitched.wind.chimes.1line
instrument.unpitched.wind.block.1line
instrument.wind.bagpipe
instrument.wind.basset.horn
instrument.wind.bassoon
instrument.wind.contrabassoon
instrument.wind.quart.bassoon
instrument.wind.quint.bassoon
instrument.wind.clarinet.a
instrument.wind.clarinet.ab
instrument.wind.clarinet.alto.eflat
instrument.wind.clarinet.alto.eflat.bassclef
instrument.wind.clarinet.bass.bflat
instrument.wind.clarinet.bass.bflat.8vb-score
instrument.wind.clarinet.bass.bflat.bassclef
instrument.wind.clarinet.bflat
instrument.wind.clarinet.c
Global constants

Contra Alto Clarinet in Eb
Contra Alto Clarinet in Eb [score sounds 8vb]
Contra Alto Clarinet in Eb [bass clef, treble transp.]
Contrabass Clarinet in Bb
Contrabass Clarinet in Bb [score sounds 15mb]
Contrabass Clarinet in Bb [bass clef, treble transp.]
Clarinet in D
Clarinet in Eb
Clarinet in G
Cor Anglais
Didgeridoo
Duduk
English Horn
Flageolet
Flute
Alto Flute
Bansuri
Bass Flute
Eb Flute
G Flute
Harmonica
Heckelphone
Mey
Nai
Oboe
Baritone Oboe
Bass Oboe
Oboe d'Amore
Ocarina
Panpipes
Piccolo
Military Piccolo in Db
Alto Recorder
Bass Recorder
Great Bass Recorder
Contrabass Recorder
Descant Recorder
Soprano Recorder
Soprano Recorder
Tenor Recorder
Treble Recorder
Alto Saxophone
Baritone Saxophone
Baritone Saxophone [score sounds 8vb]
Baritone Saxophone [bass clef, treble transp.]
Bass Saxophone
Bass Saxophone [score sounds 15mb]
Bass Saxophone [bass clef, treble transp.]
C Melody Saxophone
Contrabass (Tubax) Saxophone
Instrument.wind.clarinet.contra.alto.eflat
Instrument.wind.clarinet.contra.alto.eflat.8vb-score
Instrument.wind.clarinet.contra.alto.eflat.bassclef
Instrument.wind.clarinet.contrabass.bflat
Instrument.wind.clarinet.contrabass.bflat.15mb-score
Instrument.wind.clarinet.contrabass.bflat.bassclef
Instrument.wind.clarinet.d
Instrument.wind.clarinet.eflat
Instrument.wind.clarinet.g
Instrument.wind.coranglais
Instrument.wind.didgeridoo
Instrument.wind.duduk
Instrument.wind.englishhorn
Instrument.wind.flageolet
Instrument.wind.flute
Instrument.wind.flute.alto
Instrument.wind.flute.bansuri
Instrument.wind.flute.bass
Instrument.wind.flute.eflat
Instrument.wind.flute.g
Instrument.wind.harmonica
Instrument.wind.heckelphone
Instrument.wind.mey
Instrument.wind.nai
Instrument.wind.oboe
Instrument.wind.oboe.baritone
Instrument.wind.oboe.bass
Instrument.wind.oboe.damore
Instrument.wind.ocarina
Instrument.wind.panpipes
Instrument.wind.piccolo
Instrument.wind.piccolo.dflat
Instrument.wind.recorder.alto
Instrument.wind.recorder.bass
Instrument.wind.recorder.bass.great
Instrument.wind.recorder.contrabass
Instrument.wind.recorder.descent
Instrument.wind.recorder.soprano
Instrument.wind.recorder.soprano
Instrument.wind.recorder.tenor
Instrument.wind.recorder.treble
Instrument.wind.saxophone.alto
Instrument.wind.saxophone.baritone
Instrument.wind.saxophone.baritone.8vb-score
Instrument.wind.saxophone.baritone.bassclef
Instrument.wind.saxophone.bass
Instrument.wind.saxophone.bass.15mb-score
Instrument.wind.saxophone.bass.bassclef
Instrument.wind.saxophone.c-melody
Instrument.wind.saxophone.contrabass
Global constants

- Contrabass (Tubax) Saxophone [score sounds 15mb]
- Contrabass (Tubax) Sax [bass clef, treble transp.]
- F Mezzo Soprano Saxophone
- Soprano Saxophone
- Piccolo Saxophone in Bb [Soprillo]
- Soprano Saxophone
- C Soprano Saxophone
- Subcontrabass (Tubax) Saxophone
- Subcontrabass (Tubax) Saxophone [score sounds 15mb]
- Subcontrabass (Tubax) Sax [bass clef, treble transp.]
- Tenor Saxophone
- Tenor Saxophone [score sounds 8vb]
- Tenor Saxophone [bass clef, treble transp.]
- Woodwind
- Shakuhachi
- Tin Whistle

Beam options

For the Beam variable of NoteRest objects.

- NoBeam 1
- StartBeam 2
- ContinueBeam 3
- SingleBeam 4

Bracket types

For the AddBracket() method of BracketList objects, and the BracketType variable of Bracket objects.

- BracketFull 0
- BracketBrace 1
- BracketSub 2

Breaks

These constants are used by the SetBreakType() method of Score objects.

- MiddleOfSystem 1
- EndOfSystem 2
- MiddleOfPage 3
- EndOfPage 4
- NotEndOfSystem 5
- EndOfSystemOrPage 6
- Default 7
- SpecialPageBreak 8

These constants correspond to the menu entries in the Bars panel of the Properties window in the following way:

- MiddleOfSystem: **Middle of system.** The bar can only appear in the middle of a system, not at the end.
- EndOfSystem: No menu entry; created by Layout > Lock Format. The bar can only appear at the end of a mid-page system, not the middle of a system or the end of a page.
- MiddleOfPage: **Middle of page.** The bar can appear anywhere except at the end of a page.
- EndOfPage: **Page break.** The bar can only appear at the end of a page.
- NotEndOfSystem: No menu entry. The bar can appear anywhere except the end of a mid-page system.
Global constants

EndOfSystemOrPage  \textbf{System break}. The bar can only appear at the end of a mid-page system or the end of a page.

Default  \textbf{No break}. The bar can appear anywhere.

Note that in older versions of Manuscript the constant MiddleOfSystem was called NoBreak and the constant EndOfSystem was called SystemBreak. These older names were confusing, because they implied a correlation with the similarly-named menu items in the Properties window that was not accurate. The old names are still supported for old plug-ins, but should not be used for new plug-ins. For consistency, the old constant PageBreak has also been renamed EndOfPage, even though this did correlate correctly with the Properties window.

Accidentals
For the \texttt{Accidental} variable of Note objects.

<table>
<thead>
<tr>
<th>Accidental</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DoubleSharp</td>
<td>2</td>
</tr>
<tr>
<td>Sharp</td>
<td>1</td>
</tr>
<tr>
<td>Natural</td>
<td>0</td>
</tr>
<tr>
<td>Flat</td>
<td>-1</td>
</tr>
<tr>
<td>DoubleFlat</td>
<td>-2</td>
</tr>
</tbody>
</table>

Note Style names
For the \texttt{NoteStyle} variable of Note objects; these correspond to the noteheads available from the Notes panel of the Properties window in the manuscript papers that are supplied with Sibelius.

<table>
<thead>
<tr>
<th>Note Style</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>NormalNoteStyle</td>
<td>0</td>
</tr>
<tr>
<td>CrossNoteStyle</td>
<td>1</td>
</tr>
<tr>
<td>DiamondNoteStyle</td>
<td>2</td>
</tr>
<tr>
<td>BeatWithoutStemNoteStyle</td>
<td>3</td>
</tr>
<tr>
<td>BeatNoteStyle</td>
<td>4</td>
</tr>
<tr>
<td>CrossOrDiamondNoteStyle</td>
<td>5</td>
</tr>
<tr>
<td>BlackAndWhiteDiamondNoteStyle</td>
<td>6</td>
</tr>
<tr>
<td>HeadlessNoteStyle</td>
<td>7</td>
</tr>
<tr>
<td>StemlessNoteStyle</td>
<td>8</td>
</tr>
<tr>
<td>SilentNoteStyle</td>
<td>9</td>
</tr>
<tr>
<td>CueNoteStyle</td>
<td>10</td>
</tr>
<tr>
<td>SlashedNoteStyle</td>
<td>11</td>
</tr>
<tr>
<td>BackSlashedNoteStyle</td>
<td>12</td>
</tr>
<tr>
<td>ArrowDownNoteStyle</td>
<td>13</td>
</tr>
<tr>
<td>ArrowUpNoteStyle</td>
<td>14</td>
</tr>
<tr>
<td>InvertedTriangleNoteStyle</td>
<td>15</td>
</tr>
<tr>
<td>ShapedNote1NoteStyle</td>
<td>16</td>
</tr>
<tr>
<td>ShapedNote2NoteStyle</td>
<td>17</td>
</tr>
<tr>
<td>ShapedNote3NoteStyle</td>
<td>18</td>
</tr>
<tr>
<td>ShapedNote4StemUpNoteStyle</td>
<td>19</td>
</tr>
<tr>
<td>ShapedNote4StemDownNoteStyle</td>
<td>20</td>
</tr>
<tr>
<td>ShapedNote5NoteStyle</td>
<td>21</td>
</tr>
<tr>
<td>ShapedNote7NoteStyle</td>
<td>22</td>
</tr>
</tbody>
</table>

MuteMode constants
These are the possible values of Stave.MuteMode:

<table>
<thead>
<tr>
<th>MuteMode</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Muted</td>
<td>0</td>
</tr>
<tr>
<td>HalfMuted</td>
<td>1</td>
</tr>
<tr>
<td>NotMuted</td>
<td>2</td>
</tr>
</tbody>
</table>

Articulations
Used with \texttt{Note.GetArticulation} and \texttt{Note.SetArticulation}.

<table>
<thead>
<tr>
<th>Articulation</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Custom3Artic</td>
<td>15</td>
</tr>
<tr>
<td>TriPauseArtic</td>
<td>14</td>
</tr>
<tr>
<td>PauseArtic</td>
<td>13</td>
</tr>
<tr>
<td>SquarePauseArtic</td>
<td>12</td>
</tr>
<tr>
<td>Custom2Artic</td>
<td>11</td>
</tr>
<tr>
<td>DownBowArtic</td>
<td>10</td>
</tr>
<tr>
<td>UpBowArtic</td>
<td>9</td>
</tr>
<tr>
<td>PlusArtic</td>
<td>8</td>
</tr>
</tbody>
</table>
Global constants

- HarmonicArtic 7
- MarcatoArtic 6
- AccentArtic 5
- TenutoArtic 4
- WedgeArtic 3
- StaccatissimoArtic 2
- StaccatoArtic 1
- Custom1Artic 0

SyllableTypes for LyricItems
Used in LyricItem.

- MiddleOfWord 0
- EndOfWord 1

Accidental styles
As used by Note.AccidentalStyle.

- NormalAcc "0"
- HiddenAcc "1"
- CautionaryAcc "2"
- BracketedAcc "3"

Time signature strings
These define the unicode characters used to draw common time and alla breve time signatures, so that you can recognise these by comparison with TimeSignature.Text.

- CommonTimeString
- AllaBreveTimeString

Symbols
There are a lot of symbols in Sibelius. We've defined named constants for the indices of some of the most frequently used symbols, which can be passed to Bar.AddSymbol. For other symbols, you can work out the required index by "counting along" in the Create Symbol dialog of Sibelius, or by using the method Score.SymbolIndex. To help with the "counting along," we've defined a constant for the start of every group of symbols in the Create Symbol dialog, and these are also given below. Then for example you can access the 8va symbol as OctaveSymbols + 2.

Common symbol indices

- SegnoSymbol "1"
- CodaSymbol "2"
- RepeatBeatSymbol "5"
- RepeatBarSymbol "6"
- RepeatTwoBarsSymbol "7"
- TrillSymbol "32"
- BracketedTrillSymbol "33"
- MordentSymbol "36"
- InvertedMordentSymbol "37"
- TurnSymbol "38"
- InvertedTurnSymbol "39"
- ReversedTurnSymbol "40"
- TripleMordentSymbol "41"
- InvertedTripleMordentSymbol "42"
- PedalSymbol "48"
- PedalPSymbol "49"
Global constants

PedalUpSymbol "50"
LiftPedalSymbol "51"
HeelOneSymbol "52"
HeelTwoSymbol "53"
ToeOneSymbol "54"
ToeTwoSymbol "55"
CommaSymbol "247"
TickSymbol "248"
CaesuraSymbol "249"
ThickCaesuraSymbol "250"

Indices at the start of each group of symbols
RepeatSymbols "0"
GeneralSymbols "16"
OrnamentSymbols "32"
KeyboardSymbols "48"
ChromaticPercussionSymbols "64"
DrumPercussionSymbols "80"
MetallicPercussionSymbols "96"
OtherPercussionSymbols "112"
BeaterPercussionSymbols "128"
PercussionTechniqueSymbols "160"
GuitarSymbols "176"
ArticulationSymbols "208"
AccidentalSymbols "256"
NoteSymbols "288"
NoteheadSymbols "320"
RestSymbols "368"
ConductorSymbols "400"
ClefSymbols "416"
OctaveSymbols "448"
BreakSymbols "464"
TechniqueSymbols "480"
AccordionSymbols "496"
HandbellSymbols "528"
MiscellaneousSymbols "544"

Symbol size constants
NormalSize "0"
CueSize "1"
GraceNoteSize "2"
CueGraceNoteSize "3"
### Global constants

#### Special page break types

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>NoPageBreak</td>
<td>&quot;0&quot;</td>
</tr>
<tr>
<td>MusicRestartsAfterXPages</td>
<td>&quot;1&quot;</td>
</tr>
<tr>
<td>MusicRestartsOnNextLeftPage</td>
<td>&quot;2&quot;</td>
</tr>
<tr>
<td>MusicRestartsOnNextRightPage</td>
<td>&quot;3&quot;</td>
</tr>
</tbody>
</table>

#### Interval types

<table>
<thead>
<tr>
<th>Interval</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>IntervalDiatonic</td>
<td>&quot;-1&quot;</td>
</tr>
<tr>
<td>Interval5xDiminished</td>
<td>&quot;0&quot;</td>
</tr>
<tr>
<td>Interval4xDiminished</td>
<td>&quot;1&quot;</td>
</tr>
<tr>
<td>Interval3xDiminished</td>
<td>&quot;2&quot;</td>
</tr>
<tr>
<td>Interval2xDiminished</td>
<td>&quot;3&quot;</td>
</tr>
<tr>
<td>IntervalDiminished</td>
<td>&quot;4&quot;</td>
</tr>
<tr>
<td>IntervalMinor</td>
<td>&quot;4&quot;</td>
</tr>
<tr>
<td>IntervalMajor</td>
<td>&quot;5&quot;</td>
</tr>
<tr>
<td>IntervalPerfect</td>
<td>&quot;5&quot;</td>
</tr>
<tr>
<td>IntervalAugmented</td>
<td>&quot;6&quot;</td>
</tr>
<tr>
<td>Interval2xAugmented</td>
<td>&quot;7&quot;</td>
</tr>
<tr>
<td>Interval3xAugmented</td>
<td>&quot;8&quot;</td>
</tr>
<tr>
<td>Interval4xAugmented</td>
<td>&quot;9&quot;</td>
</tr>
<tr>
<td>Interval5xAugmented</td>
<td>&quot;10&quot;</td>
</tr>
</tbody>
</table>

#### InMultirest values

<table>
<thead>
<tr>
<th>InMultirest</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>NoMultirest</td>
<td>&quot;0&quot;</td>
</tr>
<tr>
<td>StartsMultirest</td>
<td>&quot;1&quot;</td>
</tr>
<tr>
<td>EndsMultirest</td>
<td>&quot;2&quot;</td>
</tr>
<tr>
<td>MidMultirest</td>
<td>&quot;3&quot;</td>
</tr>
</tbody>
</table>

#### Page number visibility values

<table>
<thead>
<tr>
<th>PageNumberVisibility</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PageNumberShowAll</td>
<td>&quot;0&quot;</td>
</tr>
<tr>
<td>PageNumberHideFirst</td>
<td>&quot;1&quot;</td>
</tr>
<tr>
<td>PageNumberHideAll</td>
<td>&quot;2&quot;</td>
</tr>
</tbody>
</table>

#### Page number format values

<table>
<thead>
<tr>
<th>PageNumberFormat</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PageNumberFormatNormal</td>
<td>&quot;0&quot;</td>
</tr>
<tr>
<td>PageNumberFormatRomanUpper</td>
<td>&quot;1&quot;</td>
</tr>
<tr>
<td>PageNumberFormatRomanLower</td>
<td>&quot;2&quot;</td>
</tr>
<tr>
<td>PageNumberFormatLetterLower</td>
<td>&quot;3&quot;</td>
</tr>
</tbody>
</table>

#### Special barlines

<table>
<thead>
<tr>
<th>SpecialBarline</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SpecialBarlineStartRepeat</td>
<td>&quot;0&quot;</td>
</tr>
<tr>
<td>SpecialBarlineEndRepeat</td>
<td>&quot;1&quot;</td>
</tr>
<tr>
<td>SpecialBarlineDashed</td>
<td>&quot;2&quot;</td>
</tr>
<tr>
<td>SpecialBarlineDouble</td>
<td>&quot;3&quot;</td>
</tr>
<tr>
<td>SpecialBarlineFinal</td>
<td>&quot;4&quot;</td>
</tr>
<tr>
<td>SpecialBarlineInvisible</td>
<td>&quot;5&quot;</td>
</tr>
<tr>
<td>SpecialBarlineBetweenStaves</td>
<td>&quot;6&quot;</td>
</tr>
<tr>
<td>SpecialBarlineNormal</td>
<td>&quot;7&quot;</td>
</tr>
<tr>
<td>SpecialBarlineTick</td>
<td>&quot;8&quot;</td>
</tr>
</tbody>
</table>
Global constants

Bar rest type values

- WholeBarRest: "0"
- BreveBarRest: "1"
- OneBarRepeat: "2"
- TwoBarRepeat: "3"
- FourBarRepeat: "4"

GuitarScaleDiagram type values

- ScaleTypeMajor: "0"
- ScaleTypeMinor: "1"
- ScaleTypeHarmonicMinor: "2"
- ScaleTypeMelodicMinor: "3"
- ScaleTypeDorian: "4"
- ScaleTypePhrygian: "5"
- ScaleTypeLydian: "6"
- ScaleTypeMixolydian: "7"
- ScaleTypeLocrian: "8"
- ScaleTypeWholeTone: "9"
- ScaleTypeDiminishedHalfWhole: "10"
- ScaleTypeDiminishedWholeHalf: "11"
- ScaleTypeAlteredDominant: "12"
- ScaleTypeLocrianSharp2: "13"
- ScaleTypeLydianFlat7: "14"
- ScaleTypeMajorBebop: "15"
- ScaleTypeDominantBebop: "16"
- ScaleTypeLydianSharp5: "17"
- ScaleTypePhrygianDominant: "18"
- ScaleTypeAugmentedArpeggio: "19"
- ScaleTypeMajor7thArpeggio: "20"
- ScaleType7thArpeggio: "21"
- ScaleTypeMin7Flat5Arpeggio: "22"
- ScaleTypeDiminished7thArpeggio: "23"
- ScaleTypeMajorPentatonic: "24"
- ScaleTypeMinorPentatonic: "25"
- ScaleTypeOther: "26"

FeatheredBeamType values

- FeatheredBeamNone: "0"
- FeatheredBeamAccel: "1"
- FeatheredBeamRit: "2"

Units values

- DocumentSetupUnitsmm: "0"
- DocumentSetupUnitsInches: "1"
- DocumentSetupUnitsPoints: "2"
Global constants

Orientation values
For the Orientation variable of DocumentSetup objects.

- OrientationPortrait "0"
- OrientationLandscape "1"

PageSize values
For the PageSize variable of DocumentSetup objects.

- PageSizeLetter "0"
- PageSizeTabloid "1"
- PageSizeA5 "2"
- PageSizeB5 "3"
- PageSizeA4 "4"
- PageSizeB4 "5"
- PageSizeA3 "6"
- PageSizeUSBand "7"
- PageSizeStatement "8"
- PageSizeHymn "9"
- PageSizeOctavo "10"
- PageSizeExecutive "11"
- PageSizeQuarto "12"
- PageSizeConcert "13"
- PageSizeFolio "14"
- PageSizeLegal "15"
- PageSize9_5x12_5 "16"
- PageSize10x13 "17"
- PageSizeCustom "18"

MarginType values
For the MarginType variable of DocumentSetup objects.

- PageMarginsSame "0"
- PageMarginsMirrored "1"
- PageMarginsDifferent "2"

Tuplets
These define the constants that can be passed as a style parameter to Bar.AddTuplet() and Tuplet.AddNestedTuplet().

- TupletNoNumber "0"
- TupletLeft "1"
- TupletLeftRight "2"
- TupletLeftRightNote "3"

These define the constants that can be passed as a bracket parameter:

- TupletBracketOff "0"
- TupletBracketOn "1"
- TupletBracketAuto "2"

SingleTremolos
For the SingleTremolos variable of NoteRest objects, the constants are numbers in the range 0 to 7, representing the number of tremolo beams on the stem of the note or chord. For a “z on stem” (for buzz rolls), use the value -1 or the constant ZOnStem.
**DoubleTremolo values**
For the double tremolo style variables of EngravingRules objects.

- `DoubleTremolosTouchingStems` "0"
- `DoubleTremolosBetweenStems` "1"
- `DoubleTremolosOuterTremoloTouchingStems` "2"

**Instrument name values**
For the instrument name variables of EngravingRules objects.

- `InstrumentNamesFull` "0"
- `InstrumentNamesShort` "1"
- `InstrumentNamesNone` "2"

**Types of Objects in a Bar**
The `Type` field for objects in a bar can return one of the following values:

- `Clef`
- `SpecialBarline`
- `TimeSignature`
- `KeySignature`
- `Line`
- `ArpeggioLine`
- `Bend`
- `CrescendoLine`
- `DiminuendoLine`
- `GlissandoLine`
- `OctavaLine`
- `PedalLine`
- `RepeatTimeLine`
- `Slur`
- `Trill`
- `Box`
- `BeamLine`
- `Tuplet`
- `RitardLine`
- `HighLight`
- `LyricItem`
- `Text`
- `SystemTextItem`
- `GuitarFrame`
- `GuitarScaleDiagram`
- `RehearsalMark`
- `InstrumentChange`
- `BarRest`
- `NoteRest`
- `Graphic`
- `Comment`
- `Bracket`
- `BarNumber`
- `SymbolItem`
- `SystemSymbolItem`
Global constants

What’s new in Sibelius 7

If you have used previous versions of Sibelius, you may be interested to know about the improvements to ManuScript added in Sibelius 7. The following is a list of the various new objects, methods and variables:

Dialog editor
The dialog editor for plug-in dialogs has been significantly improved, and is now available for the first time on Mac as well as Windows – see Dialog editor on page 25.

New objects
• New Accessibility object, currently with a very limited object model for controlling how Sibelius interacts with screen reading software – see Accessibility on page 45.

New methods
• New RemoveVideo() method in the Score object, to remove an attached video – see Score on page 96.
• New PrependScreenreaderText() and ScreenreaderText() methods in the Sibelius object, to override or augment Sibelius's default screen reader descriptions – see Sibelius on page 105.
• New CloseAllWindows() and CloseWindow() methods in the Sibelius objects – see Sibelius on page 105.
• New CopyOutSuffixes() method in the GuitarFrame object, replacing the deprecated Suffixes variable – see GuitarFrame on page 74.
• New ExportScoreAsPDF() and ExportPartsAsPDF() methods in the Score object – see Score on page 96.

New variables
• BarObject objects now provide a variable for the alpha channel of their color – see BarObject on page 54.
• Folder objects now provide a variable for the total number of files of any file type in the folder – see Folder on page 73.
• InstrumentType objects now provide a variable returning the style ID of the tab instrument type used to determine the tuning of chord diagrams on standard notation staves – see InstrumentType on page 82.
• NoteRest objects now provide a variable for the stem direction of the note – see NoteRest on page 87.